

Graphics for the IBM PC

Dan Illowsky
Michael Abrash



GRAPHICS FOR THE IBM PC



Dan Illowsky and Michael Abrash are the authors of several popular arcade-style games, including *Snack Attack*, *County Fair*, *Cosmic Crusader*, and *Big Top* for the IBM PC and Apple II microcomputers. They have also coauthored several articles for computer magazines. Both are officers of Funtastic, Incorporated, a software publishing house located near Philadelphia; in addition, they recently formed a computer consulting firm, Mida Corporation.

GRAPHICS FOR THE IBM PC

by

Dan Illowsky and Michael Abrash

Howard W. Sams & Co., Inc.

4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

Copyright © 1984 by Dan Illowsky and Michael
Abrash

FIRST EDITION
FIRST PRINTING—1984

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-22191-8
Library of Congress Catalog Card Number: 83-51617

Edited by *Katherine Stuart Ewing*

Printed in the United States of America.

IBM is a trademark of International Business
Machines Corporation

PREFACE

This is as much a “why to” book as a “how to” book. By leading you through a hands-on tutorial, we will develop your ability to design your own graphics programs on the IBM PC. The emphasis is on improving your ability to pursue your own ideas. Rather than just showing you pretty pictures, we provide the tools needed to create pictures, games, charts, diagrams, and artwork of all kinds. Rather than present you with dozens of program listings too long to type, we take you through each graphics command with both short “hands-on” examples and carefully designed, thoroughly explained sample programs. We also take you through the process of designing two arcade-style games, two graphing programs, and a character-generator program that lets you define your own fonts. You will understand how each program works and, more importantly, how to apply this knowledge to developing your own programs.

Advanced BASIC on the IBM Personal Computer is a powerful graphics language, vastly superior to earlier microcomputer BASICs. Advanced BASIC makes sophisticated graphics readily available to the average programmer for the first time. This book is designed to be a key to your growth in tapping the power of your PC. We will give you a comprehensive knowledge of the tools available, as well as a solid understanding of good program design. If you are already familiar with BASIC and the IBM PC, this book is an excellent complement to your skills. It fills in the missing details and plugs the gaps in your knowledge.

If you are not familiar with BASIC or the PC, this book will get you started and, as an extra benefit, provide you with a lot of useful experience with BASIC language in general.

We expect you will take this book off the shelf time and again for reference use. Much of the information we have gathered in two years of working with the PC is presented in detail, including useful applications, undocumented capabilities, and traps to avoid.

One of the great strengths of the PC is that all the fundamental graphics functions are part of the language. The programmer need not waste time programming basics, such as dots, lines, and circles. This allows the programmer to spend time designing the programs, leaving the imagination free to wander. We will familiarize you with the graphics tools that BASIC makes available, and get you started applying them—the imagination is up to you!

There are many fine books about specific graphics techniques, such as plotting three-dimensional surfaces, or windowing, clipping, and zooming, but this book is not one of them. These topics require an entire book simply for a proper introduction and are much more appropriate for high-powered mainframe computers than for microcomputers. The IBM PC is ideal for producing colorful, detailed displays without requiring that the programmer invest years in special training.

There's not a great deal that you need to know before we start. You do need an IBM PC with a Color/Graphics Adapter (more about this in Chapter 1) and a disk drive. Familiarity with the PC and BASIC is useful, but not required. A willingness to participate by entering the examples is essential for you to benefit properly from this book.

Let's get going—and welcome to the world of PC graphics!

DAN ILLOWSKY AND MICHAEL ABRASH

We thank Mark Karaman and Morris Blackman for their photographic assistance. We also express our sincere gratitude to Lisa Stahr of *PC World* magazine for helping us get our feet wet in technical writing. And, of course, we extend special thanks to Barbara and Shay.

CONTENTS

PART 1 — AROUND THE PC

CHAPTER 1

GETTING STARTED WITH YOUR PC	16
Required Equipment — And . . . Action — Creating Your BASIC Disk — Onward to BASIC	

CHAPTER 2

GETTING STARTED WITH BASIC	33
Getting onto the Graphics Screen — A Simple Test — Concepts of BASIC — Statements, Functions and Commands — The Cursor — Some Other Terms — Editing— Other Special BASIC Keys — Multiple Com- mand Lines — Comments and Good Programming Style — A Kaleidoscope Program — Errors — Advanced BASIC — The Tutorial	

PART 2 — A HANDS-ON GRAPHICS TUTORIAL

CHAPTER 3

INTRODUCTION TO THE TUTORIAL	52
Before We Begin	

CHAPTER 4

MEDIUM-RESOLUTION GRAPHICS—THE PSET STATEMENT	55
Some Useful Background — Setting Up Medium-Reso- lution Graphics — More Useful Background — PSET and PRESET— Text in the Graphics Screen — Rela- tive Addressing: Using the Last Point Referenced	

CHAPTER 5

THE POINT FUNCTION	75
An Example of the POINT Function	

CHAPTER 6

THE LINE STATEMENT	79
An Example of the LINE Statement	

CHAPTER 7

ELLIPSES, ARCS, AND WEDGES—THE CIRCLE STATEMENT ...	84
Arcs and Wedges — Aspect and the Screen	

CHAPTER 8

THE PAINT STATEMENT—THE ARTIST'S BRUSH	93
Painting Tips	

CHAPTER 9

A PIE CHART PROGRAM	98
Improving the Pie Chart Program	

CHAPTER 10

ANIMATION FROM BASIC—THE GET AND PUT STATEMENTS .	103
The Five PUT Options — Animation — Fun and Games	

CHAPTER 11

BLOCKBUSTER—AN ARCADE-STYLE GAME	113
Game Design — Blockbuster — Programming the Game — The Ball's in Your Court	

CHAPTER 12

THE DRAW STATEMENT—A LANGUAGE WITHIN A LANGUAGE	133
Moving Around the Screen—The M Subcommand — Moving Around the Screen—The Movement Subcommands — Controlling the Pen — The Color Subcommand — The Scale Subcommand — The Angle Subcommand — Variables in the DRAW Statement — DRAW Substrings — Error Conditions — Summary of the DRAW Statement	

CHAPTER 13

A CHARACTER GENERATION PACKAGE	151
The Nature of Text — Subroutines — Structure of the Character Generator — The Initialization Subroutine — The Character Drawing Subroutine — Using the Character Generation Package — Inserting the Package in Your Programs — Summary of the Character-Generation Package	

CHAPTER 14

HIGH-RESOLUTION GRAPHICS MODE	171
High-Resolution Mode — High Resolution versus Medium Resolution	

CHAPTER 15

A FUNCTION-GRAPHING PROGRAM	179
A Graphing Package — The Program — Summary and Enhancements	

CHAPTER 16

TEXT-MODE GRAPHICS	189
Setting Up the Text-Mode Screen — Using Text Mode — The SCREEN Function — Multiple Screen Pages — Text-Mode Graphics	

CHAPTER 17

RACECAR—AN ARCADE-STYLE GAME	214
Racecar — Programming Racecar — Improving Racecar	

CHAPTER 18

SUMMARY OF THE TUTORIAL	232
Where Do You Go from Here?	

PART 3 — ADVANCED TOPICS

CHAPTER 19

A GRAB BAG OF GRAPHICS TRICKS	236
Color in High-Resolution Mode — Artifacts — More on Color — Colored Text in Graphics Mode — The Third Palette — Checking the Screens — Saving the Screen to Disk—BSAVE and BLOAD — Printing the Screen — The Keyboard — Clearing Excess Keys — Entering All 255 Characters — The Full PC Character Set — Aligning the Screen — Variable Scroll Window — Fill Your Own Grab Bag	

CHAPTER 20

INSIDE PC GRAPHICS	264
Some Useful Terms — Memory-Mapped Video — Accessing Memory from BASIC — Screen Memory Organization — Text Mode — Screen Memory Organization — High-Resolution Mode — Screen Memory Organization — Medium-Resolution Mode — Why PEEK and POKE?	

APPENDIX A

THE SET OF CHARACTERS AVAILABLE FROM BASIC	281
--	-----

APPENDIX B

THE FULL 255 CHARACTER SET OF THE IBM PC	286
--	-----

APPENDIX C

DECIMAL, HEXADECIMAL, AND BINARY CONVERSION TABLE	289
--	-----

GLOSSARY	293
----------------	-----

INDEX	307
-------------	-----

A NOTE TO THE READER

The programs in this book were not written as applications software but as educational examples of what your personal computer can do. All of the programs have been tested and work on the machine configuration for which they were designed. The programs, or subroutines, are unprotected. This means that you can modify them to better understand how they work or to fit a different machine configuration.

What Is a Combo Pack?

A Combo Pack, like this package, is a step beyond your average technical book. While most books give you programming examples through printed listings (which we do here), Combo Packs provide the book and the listings recorded on magnetic media, either diskette, cassette tape, or both.

Every effort has been made to be clear, concise, and informative about how these programs and routines work. If you experience any difficulty with the software operations, the solution can be found in the book or in your computer manuals.

We are rather proud of the time and effort that went into preparing the Combo Pack. If you have purchased the Combo Pack and have enjoyed using it, let us know your thoughts. Your comments will be valuable in preparing future Combo Packs.

LOADING INSTRUCTIONS

If you bought this book as part of a Combo Pack, a disk is included. Instructions for using the disk follow.

You will need to create a system disk with DOS and BASICA on it. Neither DOS or BASICA is included with this disk. If you do not have them, they are available from your IBM dealer.

The following instructions for creating a system disk assume that your computer has two disk drives. If it only

has one, refer to the IBM DOS manual for information specific to your hardware configuration.

1. Insert your IBM DOS System Disk in drive A (left-hand drive), then turn on your computer or reboot it by pressing **Ctrl-Alt-Del**. Insert a blank disk in drive B (right-hand drive). Type **FORMAT B:/S** and press Enter. You will then be prompted to ensure that you have the proper disks in the drives. Press Enter and the disk in drive B will be formatted.
2. When the disk is formatted, type **COPY BASICA.COM B:** and press Enter. This will copy BASICA to the disk in drive B.
3. Remove the IBM DOS System Disk from drive A.
4. Insert the Combo Pack Disk in drive A.
5. Type **COPY *.* B:** and press Enter. This will copy all of the program files from the Combo Pack disk to the disk in drive B.
6. Remove the Combo Pack disk from drive A and put it in a safe place.
7. Label the disk in drive B as your new Combo Pack master disk.

You have created a system disk including the Combo Pack programs. You can use this disk to boot from, and then run BASICA. At that point, you can load and work with any of the files on the disk.

PART 1

AROUND THE PC

CHAPTER 1

GETTING STARTED WITH YOUR PC

In the next two chapters, we will provide you with the background necessary to start the actual tutorial on graphics. You should feel free to skip any information in either of these chapters with which you already feel comfortable. Chapter 3 begins the actual graphics tutorial. In Chapter 1, we will explain the equipment and software you will need, how to start up your IBM PC, and how to make your own disk to use with this book.

We will teach you quite a bit about BASIC in general in the course of providing you with enough information to start the graphics tutorial, and further aspects of BASIC will be discussed as needed in the following chapters. If you have no previous experience with BASIC, IBM's BASIC manual, which was included when you purchased your computer, is an excellent additional reference. Refer to it any time you are uncertain or simply want more information about the BASIC language.

REQUIRED EQUIPMENT

There are many possible equipment combinations for the IBM Personal Computer, also known as the PC. There is a minimum configuration required for this tutorial. A system unit, keyboard, 64K of memory, the PC-DOS operating system, and Advanced BASIC are necessary. Also

needed is a plug-in board called the Color/Graphics Adapter, and therein lies a tale.

Dual Screens

Most microcomputers have a single screen on which both text and graphics can be displayed. The screen may have several different modes, but there is only the single screen.

IBM took a different approach. IBM wanted a computer that produced the sharpest, clearest text possible to attract the word-processing and business market. IBM also wanted a computer capable of color graphics as good as anything available in the PC's price range. Rather than compromise on a screen which excelled at neither text display nor graphics, IBM chose to offer the buyer the choice of either a text screen or a graphics screen (or both).

The text screen is IBM's *monochrome screen* (Fig. 1-1). It has a sharp green display and no graphics capabilities. The screen is driven by an add-on board called the *Monochrome Adapter*, which plugs into one of the five available expansion slots inside the PC. There are also a few screens now available from manufacturers other than IBM that work with the Monochrome Adapter. Any screen driven by the Monochrome Adapter cannot produce graphics.

The graphics screen may be a television screen, a black and white monitor, or a color monitor. Any of these screens can be driven by the *Color/Graphics Adapter* (Fig. 1-2), an add-on board which, like the Monochrome Adapter, plugs into an expansion slot inside the PC. In fact, about the only screen the Color/Graphics Adapter can't drive is IBM's monochrome screen. The graphics screen can display either text or graphics; however, the text displayed is not of the quality of that produced on the monochrome screen. The Color/Graphics Adapter can produce up to 16 colors or can draw up to 200 rows by 640 columns of dots.

Because this tutorial is about graphics, it is essential that you have a Color/Graphics Adapter installed in your PC, although you may have the Monochrome Adapter as well. Because graphics are so important to this tutorial, we

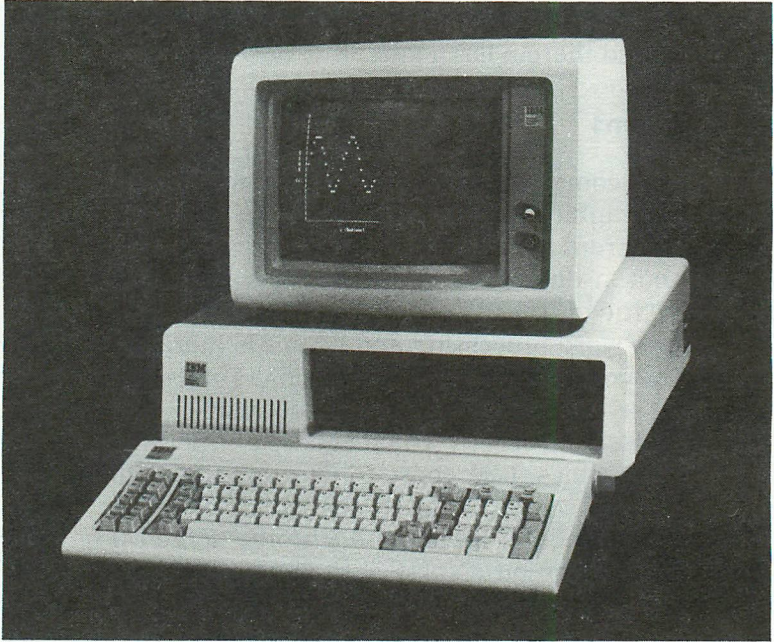


Fig. 1-1. Monochrome display.

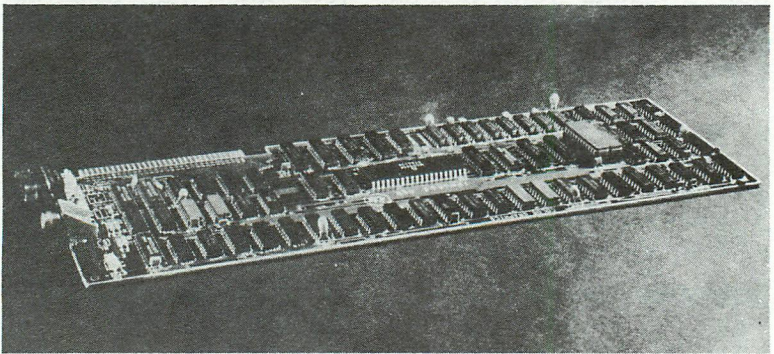


Fig. 1-2. Color/Graphics Adapter card.

will take a moment here to discuss the various screens that may be used with the Color/Graphics Adapter.

Color/Graphics Screens

The quality of graphics possible with the Color/Graphics Adapter varies greatly with the type of screen. The follow-

ing discussion is intended to serve two purposes. First, if you are planning to purchase a Color/Graphics Adapter, our description should help you match a screen to your needs. Second, the results you get from lower-quality screens can be quite different from what you might expect, particularly when color is involved.

The screens used with the Color/Graphics Adapter fall into three groups: composite monitors, RGB monitors, and televisions. *Composite* monitors (Fig. 1-3) often have black and white (or green or amber), as well as multicolored, screens. Essentially a tv with the channel changer removed, these monitors must decode one signal into the three necessary color signals before producing a picture. The better composite monitors include features such as antiglare screens and contrast control.



Fig. 1-3. Composite monitor.

An RGB monitor (Fig. 1-4) accepts red, green, and blue color signals directly (hence the name RGB). RGB moni-

tors are typically constructed to higher standards than are composite monitors, and provide the highest-quality picture with the fewest problems. RGB monitors are also correspondingly expensive.

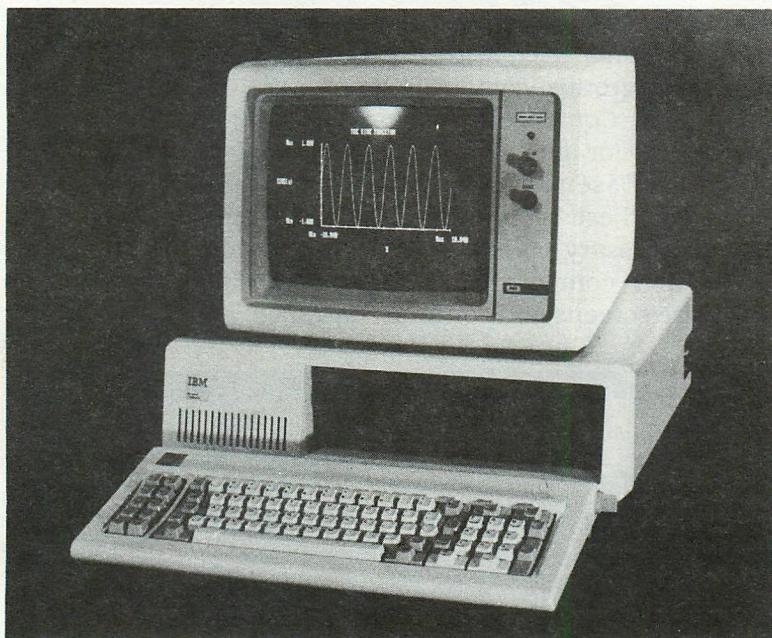


Fig. 1-4. RGB monitor.

A *television set* (Fig. 1-5) accepts a signal through an antenna, and then must decode the signal twice before producing a picture. Televisions are designed to display program broadcasts rather than computer output, and, while they are adequate for graphics, they are barely acceptable for text work (even when only 40 columns of text are displayed). A television is not a good choice for the only monitor in a system.

To produce both graphics and extensive text work, either a high-quality RGB monitor or a monochrome screen is a better choice than a tv. A standard black and white monitor, such as one of those made by NEC and Sanyo, is an excellent choice for those on a budget (Fig. 1-

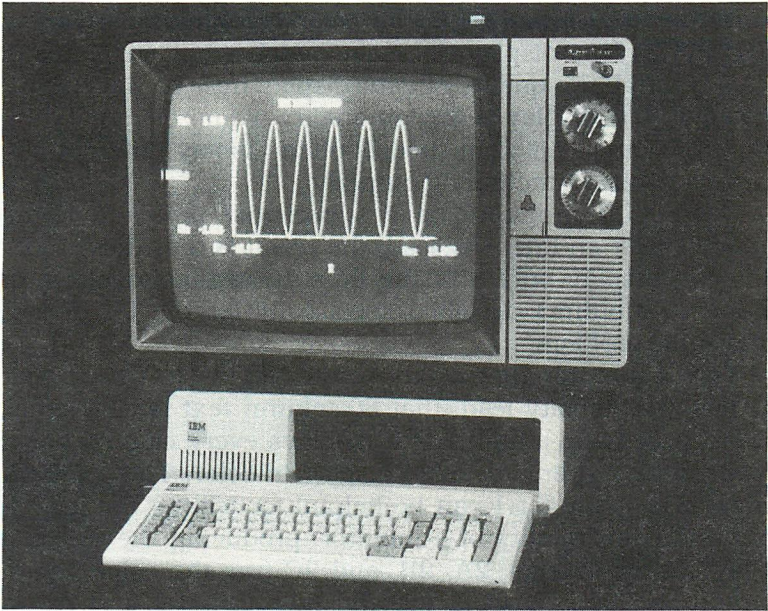


Fig. 1-5. Television used as IBM PC monitor.

3). Such a monitor is more than adequate for text work and can also display graphics, albeit not in color. Together the Color/Graphics Adapter and a monitor cost about \$500; for less than the price of the Monochrome Adapter and screen, you can have both graphics and text capability. One last option is to use any of the displays in conjunction with IBM's monochrome screen; in this case, the text quality of the color screen is immaterial because most text work will take place on the monochrome screen.

Figs. 1A, 1B, and 1C (see the color photograph section) show the three types of displays. (The programs which produce the images shown in Figs. 1A, 1B, and 1C are not available in this book.) Color, resolution, and clarity really do vary widely among RGB monitors, composite monitors, and televisions.

There are potential problems with each of the types of displays. Televisions and composite monitors tend to be able to display only every other column of colored graphics. Only RGB monitors display 80-column text well, and

only on RGB monitors do all color combinations display legibly. If your display does not match what we claim in this tutorial, the quality of your display may be the culprit. Even if your display is of high quality, your colors may differ from ours—different models often display different colors even when functioning properly.

Even RGB monitors have a potential problem. Some models lack an intensity lead; the result is that these models can display only 8 of the PC's 16 available colors. In general, monitors designed specifically for the PC do not have this problem.

Test a monitor thoroughly before buying it. Check that it has an intensity lead, that 80-column text is legible (for word processing), and that it displays every colored dot if your work requires detailed display in color. The program shown in Listing 1-1 can be helpful for testing the legibility and color quality of a monitor, producing the test display shown in Figs. 2A, 2B, and 2C (see the color photograph section). (We will tell you how to enter and run a program shortly.)

Each monitor is attached in a different fashion. RGB monitors attach to the multipin connector on the back of the Color/Graphics Adapter (Fig. 1-6). Composite monitors attach to the jack just above the RGB connector (Fig. 1-6). Televisions require an additional piece of equipment called an *RF modulator*. The RF modulator attaches to the Color/Graphics Adapter inside the PC (your computer store will probably help you install it), and sits on the back side of the computer (Fig. 1-6). A cable leads from the RF modulator to a connection box on the tv (Fig. 1-7); the cable should be looped around an iron ring as shown to reduce interference. The connection box has a switch that lets the user select between normal television stations and the computer display. This switch should always be in the "computer" position when the PC is displaying something on the tv screen. An RF modulator typically costs about thirty dollars.

Other Color/Graphics Adapters

Several companies besides IBM now produce Color/Graph-

Listing 1-1. Color Text Modes.

```

100 REM A program that demonstrates the color text modes.
110 REM 80 character text mode may be illegible on TV's
120 REM and low-quality monitors. This program is useful
130 REM for testing the quality of a color monitor.
140 SCREEN 0,1:COLOR ,0:KEY OFF:CLS 'Set COLOR TEXT MODE
150 DIM A$(40) 'Set aside memory
160 A$="40 CHARACTER COLOR TEXT MODE"
170 WIDTH 40 'Set to 40 characters on line
180 LOCATE ,,0 'Turn cursor off
190 X=5 'Set printing location
200 GOSUB 290 'Display text on screen
210 A$="80 CHARACTER COLOR TEXT MODE"
220 COLOR ,0:WIDTH 80 '80 columns/black background
230 LOCATE ,,0 'Turn cursor off
240 X=25 'Set printing location
250 GOSUB 290 'Display text on screen
260 WIDTH 40 'Set back to 40 columns
270 COLOR 7,0,0:KEY ON:CLS 'Restore screen
280 END 'Return control to BASIC
290 FOR BG=0 TO 7 'Each line is printed in
300 LOCATE 5+BG,X ' a different color
310 FOR FG=1 TO LEN(A$) 'Characters will be 16
320 COLOR FG MOD 32,BG,4 ' different characters
330 PRINT MID$(A$,FG,1); 'Display the next character
340 NEXT FG 'Display characters until
350 PRINT ' done displaying all
360 NEXT BG ' eight lines
370 COLOR 7,0:LOCATE 18,X:PRINT "PRESS ANY KEY TO CONTINUE"
380 A$=INKEY$:IF A$="" THEN 380 'Wait for key to continue
390 RETURN 'Finished with this string

```

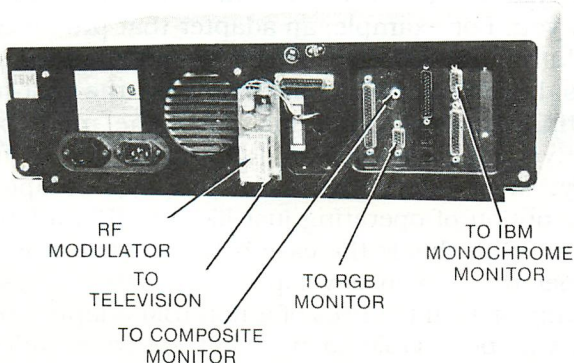


Fig. 1-6. Display connectors.

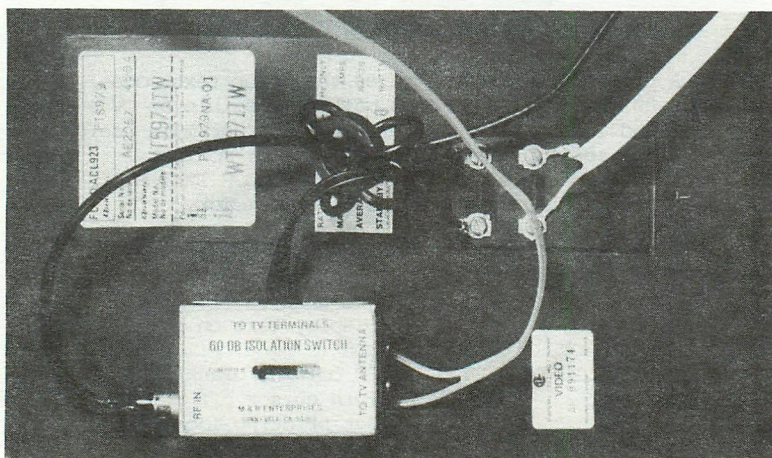


Fig. 1-7. Connection box.

ics Adapters, and the number is growing. Some of these alternative boards provide more colors, or higher resolution, than IBM's board, while some have serial or parallel ports, extra memory, or monochrome capability as well as the Color/Graphics Adapter circuitry. All provide something that the IBM adapter does not, and the alternative boards often save money, expansion slots, or both.

The single most important point to consider when buying a non-IBM Color/Graphics Adapter is how much like the IBM adapter it works and how important that similarity is to you. For example, an adapter that produces only a 640 column by 400 row screen would produce detailed pictures, but would not properly support the vast majority of existing graphics programs. Also, the more powerful alternative adapters require RGB monitors to function properly. Most alternative Color/Graphics Adapters provide the option of operating just like the IBM adapter, but make sure that this is the case before purchasing. Finally, remember that if you develop software that takes advantage of any special features of a non-IBM adapter, other PC owners will be unable to use that software unless they have the same adapter you do.

The Keyboard

There are several other pieces of equipment that you should be familiar with before we begin using the PC. Let's start with the keyboard.

The keyboard (Fig. 1-8) is discussed in both the *DOS* and *BASIC* manuals. There are a few features that we will point out now, as we will be working with them later.

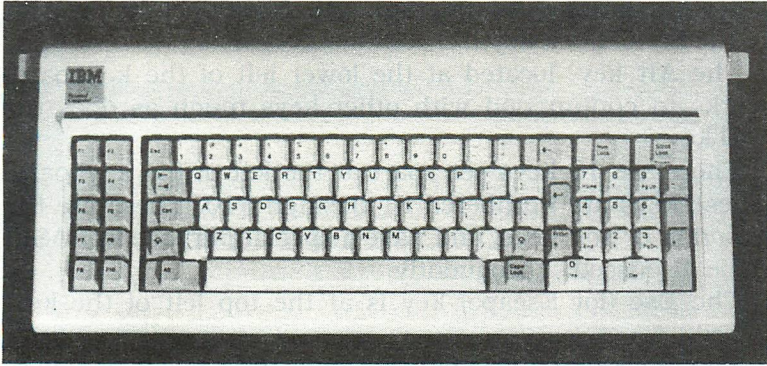


Fig. 1-8. The IBM PC keyboard.

The most important key is the **Enter** key. This is the double-height gray key above the **PrtSc** key, with an arrow angled down and to the left on it. The Enter key is used to send whole command lines to the computer and will be referred to frequently.

Incidentally, the PC counts everything you type before you press Enter as one line. Typed lines can reach the right margin and wrap around to the next line down particularly when the screen is only 40 columns wide. Occasionally (for example, when running the DISKCOPY program), the computer acts after receiving a single character, but, in general, the PC does nothing until the Enter key has been pressed. The Enter key is analogous to the Return or Newline keys on other computers.

Spaces are important in command lines, so type them, too. We often omit commas and periods, so the lines printed in this book appear precisely as they are to be typed.

A control key is entered by holding down the **Ctrl** key and pressing the specified key once. (The Ctrl key is at the middle left of the keyboard.) For example, Ctrl-C means to hold the Ctrl key down while pressing the C key once. There are two special control keys. **Ctrl-Break** will stop any BASIC program. (The Break key is at the top right of the keyboard, and is also labeled Scroll Lock.) **Ctrl-Num Lock** will suspend any program, and pressing any other key will continue the program. (The Num Lock key is next to the Scroll Lock key.)

The **Alt** key, located at the lower left of the keyboard, works in conjunction with other keys much as does the Ctrl key.

There are 10 keys set apart at the left of the keyboard. These keys are numbered F1 through F10. These are the *function* or *soft* keys, and have a special purpose in BASIC to be discussed subsequently.

The **Esc** (for Escape) key is at the top left of the keyboard.

The **Backspace** key (labeled only with a left arrow) is directly to the left of the Num Lock key. This key erases the last character typed.

The set of 11 white keys at the far right is a dual-purpose keypad. Normally, these keys perform functions such as moving the cursor around the screen, inserting, and deleting. If, however, the Num Lock key is pressed, the keys become numeric and send the numbers shown on them. Thus the **Home** key can send the number 7. If the Num Lock key is pressed again, the keypad returns to its original state.

One special key combination is **Ctrl-Alt-Del** (pressing the Ctrl, Alt, and Del keys simultaneously). This causes the PC to completely restart, and will get you out of almost any jam. However, you will lose whatever you were working on, so use this with care. Incidentally, switching the PC off, waiting 10 seconds (or until you hear the internal fan stop), and switching the power back on will get you out of any jam, but again you will lose any work in progress.

Disk Drives

Disk drives are used to store the equivalent of up to 360,000 characters of programs and data on floppy disks (Fig. 1-9). The use of disk drives is described in the *DOS* manual. Our only caution is to emphasize that you should never touch those parts of the disks exposed through the jacket cutouts. Always keep the disk in its protective jacket when it is not in the drive. Disks are not designed for strength, occasionally wear out, and are vulnerable to smoke, coffee, and fingerprints, so you must be careful—and you absolutely *must* keep at least one backup of any important disks. The 30 seconds of your time required to back up a disk is nothing compared to the time required to recreate lost information. The disk copying procedure is detailed in the *DOS* manual.

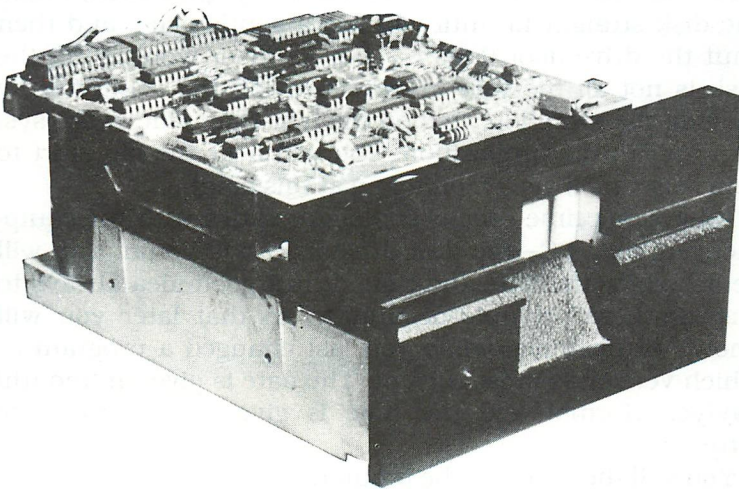


Fig. 1-9. Floppy disk drive.

Note that the term “disk” can refer to either the disk drive or to the floppy disk that you insert into the disk drive. It is the floppy disk that contains information.

Software

PC-DOS (which we will simply call “DOS” from now on) and Advanced BASIC are all the software required for this

tutorial. Both programs are on the disk that came with your *DOS* manual. You should never use master disks such as the *DOS* disk that came with your PC; if you have not already done so, make several copies of your *DOS* disk, as described in the *DOS* manual.

AND . . . ACTION

We can now begin to use the PC. First, open the door on drive A: (the left-hand drive) by lifting up gently. Then place a working copy of the *DOS* disk in drive A. The label on the disk should be facing up, and the edge with the oval cutout in it should be the first edge into the computer. Your fingers should be well away from any exposed surfaces (Fig. 1-10). (This process is covered in detail in the *DOS* manual—refer to it if you have any questions.) Insert the disk straight in until you feel a gentle click, and then shut the drive door (Fig. 1-11). If the door won't shut, the disk is not all the way in. Now turn on the computer by moving the switch located on the rear right side of the system unit to the up position. You will hear the fan start to spin. Also, turn your monitor on at this point.

After some time—the exact length varies with the equipment in your PC—the disk drive light will go on. You will be asked for the date and time. It is a good idea always to answer these prompts accurately, so that later you will know, for example, when you last changed a program or which version is most current. The date is given in month/day/year format and the time is given in hour:minute form.

You will then receive the prompt:

A>

which indicates that *DOS* is waiting for you. You have successfully started your computer. This process is known as *booting* the computer. If you do not get the A> prompt, consult your *DOS* manual, which contains complete instructions on starting *DOS*. One note: if you have both the Monochrome and Color/Graphics Adapters, your system will most likely boot with the initial display on the

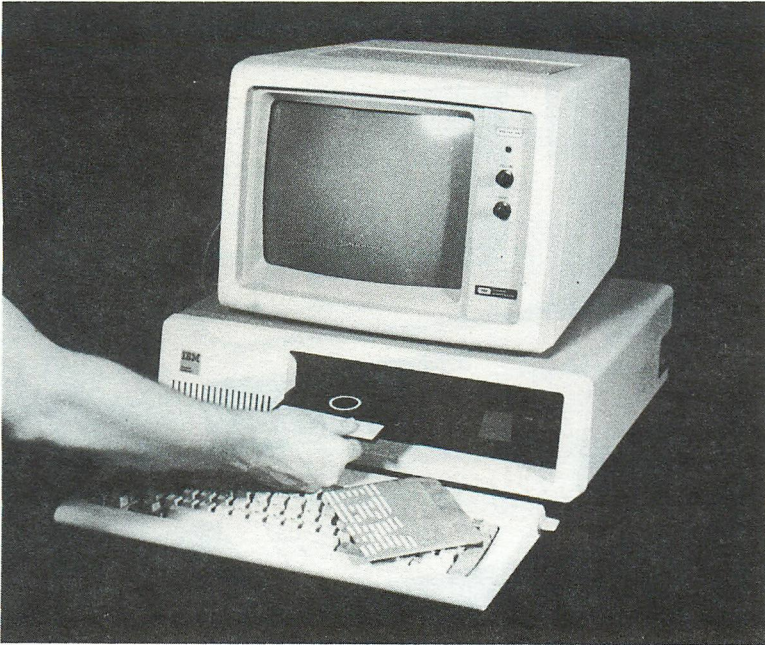


Fig. 1-10. Holding the disk.

monochrome monitor. We will tell you how to switch monitors shortly.

DOS provides you with an environment in which you can run programs that are stored on a disk. This is done by typing the name of the program and pressing Enter in response to the `A>` prompt. Advanced BASIC is the program `BASICA.COM` on the DOS disk. To run Advanced BASIC, type:

BASICA and press Enter

in response to the `A>` prompt. Try this now. In DOS it doesn't matter whether you use uppercase, lowercase, or a mixture of the two.

The screen will clear, and you will receive Advanced BASIC's sign-on message which includes: the version of BASIC, the amount of memory free, and the standard BASIC prompt **Ok**. If you have 128K or more memory, there will be about 60,000 bytes free. BASIC can never

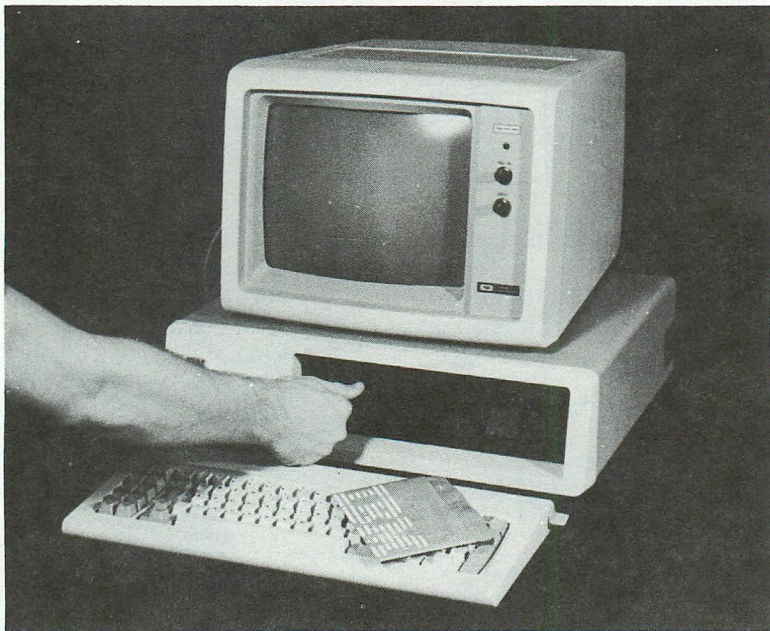


Fig. 1-11. Disk insertion.

have more room available than this, no matter how much memory you have. If you have less than 128K of memory, your BASIC workspace will be smaller than the 60,000 maximum. This will make no difference with regard to our tutorial, but it will limit the maximum size of BASIC programs you can run.

Note that whenever we say "BASIC" from now on, we mean *Advanced BASIC*, that is, the program `BASICA.COM` rather than the more limited `BASIC.COM`.

By and large, BASIC is self-contained, and we will concentrate on BASIC rather than DOS; still, there are a few DOS commands that are particularly useful. `DISKCOPY` enables you to back up disks; `FORMAT` initializes a blank disk so that your files can be saved on it; `CHKDSK` lets you know how much room is left on a disk; and `DIR` lists the names and sizes of the files on a disk. Any of these commands may be entered in response to the `A>` prompt. Each command is discussed in detail in the *DOS manual*;

be sure to study the manual before using any DOS command.

CREATING YOUR BASIC DISK

We don't need most of the programs on the DOS disk for our BASIC work. Because there is not a large amount of space left on the DOS disk, it makes sense for us to make a separate disk customized to meet our needs.

Make sure the DOS disk is in the left-hand drive. If you are in BASIC, type the command:

SYSTEM and press Enter

When you see the A> prompt, you are ready to begin. If you have trouble, just switch off the computer, wait a few seconds, and turn the computer back on. This will get you out of any jam, but you may lose work in progress.

You will need a blank disk. We suggest you put a write-protect tab over the write-enable notch on the DOS disk. This will guarantee that you cannot "wipe out" the contents of that disk. With the DOS disk in drive A: (the left-hand drive), type:

FORMAT A:/S and press Enter

When the computer prompts you to press any key, remove the DOS disk and put the blank disk in drive A:, then press a key. When the disk is formatted, the monitor will show a message indicating how much room there is on the disk. This will be 160,256 or 322,560, depending on whether you have single- or double-sided disk drives. Jot down this number for later reference. Then type the character N in response to the prompt:

FORMAT ANOTHER (Y/N)

You will see the A> prompt on your monitor.

Now copy the BASICA.COM program onto your new BASIC disk. If you have two disk drives, put the DOS disk in drive B: and type:

COPY B:BASICA.COM A:

If you have one drive, type:

```
COPY B:BASICA.COM A:
```

and follow the prompts on the screen. (The DOS disk is the source disk.)

Now label the new disk as your graphics tutorial workspace. All of our work will take place on this disk. To start sessions from now on, just boot this disk, type:

```
BASICA
```

in response to the A> prompt, and switch screens if necessary, as discussed in the next chapter.

If you have single-sided disk drives, you will run out of space during the tutorial if you save all the files you type. (You have a single-sided drive if the PC printed 160,256 for the amount of disk space available on the disk previously mentioned.) When you run out of space, you will receive one of two error messages: "Too many files" or "Full disk". If this happens, format another disk, label it as your graphics tutorial workspace disk No. 2, and use it from then on. You will have to retype what you had attempted to save. Alternatively, you could use the KILL command to make some room on the disk, but because you may want to refer to the programs later, you might as well keep the programs and use a second disk.

You can use the DOS command CHKDSK to see how much room is left on the disk, and head trouble off at the pass by switching to a second disk before you have to. The DOS manual contains information on the CHKDSK command.

If you have a hard disk, you can use it if you wish, but in order to keep your tutorial files separate, and to avoid the possibility of accidentally affecting other, nontutorial files, we suggest you use a floppy disk for this tutorial.

ONWARD TO BASIC

You now know how to set up and boot your PC and how to use some of the features of DOS. It's time to begin using BASIC.

CHAPTER 2

GETTING STARTED WITH BASIC

In this chapter we will tell you how to start BASIC, how to make sure you are on the correct screen, and how to create, run, and save BASIC programs. We will also cover fundamental concepts of BASIC, as well as details about using BASIC such as program modification and error handling. Finally, we will give you a sample of what your PC can do.

Note that this book was designed for BASIC 1.1. If you are using another version of BASIC, you may notice some slight differences.

GETTING ONTO THE GRAPHICS SCREEN

To start Advanced BASIC, boot the BASIC disk you made in Chapter 1, then type:

BASICA and press Enter

in response to the **A>** prompt. For those of you with only the Color/Graphics Adapter, that's all there is to starting BASIC. However, those of you with both the Monochrome Adapter and the Color/Graphics Adapter have some work to do. If you have only the Color/Graphics Adapter screen, skip to the "A SIMPLE TEST" heading further on in this chapter.

If you have both adapters, and the **A>** prompt appears on the monochrome screen, then you must run a BASIC program to switch to the graphics screen because graphics cannot be done on the monochrome screen. If you have

both adapters, odds are that your PC will, in fact, boot on the monochrome screen. You can boot on the graphics screen by resetting the switches inside the PC if you wish; consult IBM's *Guide to Operations* manual for more details.

To switch screens, you must first enter a BASIC program, named *Switch*, that will switch screens, then save the program to disk so you can readily run it each time you start BASIC. We will take you through this process step by step.

First, start Advanced BASIC by typing:

BASICA and pressing Enter

in response to the A> prompt. If you are using a Combo Pack disk, load the program by typing:

LOAD "LZ-1"

Then skip to the **RUN** command described on the next page. Load any of the listings in this book by using **LOAD** and the listing number. Then type in the program shown in Listing 2-1 exactly as it appears. (If your screen displays 40 columns of text, some lines will wrap when they reach the right margin. Keep typing. Don't press Enter until the line is complete.) At the end of each line, press the Enter key. You can check what you have typed by typing the command:

LIST and pressing Enter

If you make a mistake, just retype the line properly. If you have a great deal of trouble, type:

NEW and press Enter

to clear your previous work, and retype the entire program.

Once you are certain the program is typed correctly, type the command:

RUN and press Enter

Listing 2-1. Switch.

```

100 REM Program to switch from color/graphics adapter
110 REM to monochrome adapter or vice-versa.
120 REM Figure which adapter is active & switch to other
130 LOCATE 1,1,0:DEF SEG=0:A=(PEEK(1040) AND 48):
    IF A<>48 GOTO 210
140 REM Switch to color/graphics adapter
150 KEY OFF:CLS
160 A=PEEK(1040):POKE 1040,(A AND 207) OR 32
170 SCREEN 0,0:COLOR 7,0:LOCATE 1,1,1,6,7
180 KEY ON:WIDTH 40
190 END
200 REM Switch to monochrome adapter
210 KEY OFF:CLS
220 A=PEEK(1040):POKE 1040,A OR 48
230 SCREEN 0,0:COLOR 7,0:LOCATE 1,1,1,12,13
240 KEY ON:WIDTH 80
250 END

```

The monochrome screen will blink off, the graphics screen will go blank, and the Ok prompt will appear on the graphics screen. Type:

RUN and press Enter

again, and you will find yourself back on the monochrome screen. *Switch* senses which screen you are on and switches to the other screen. If the program fails to work, reboot the system, restart BASIC, and retype the program *Switch* as described above. If you have problems, don't despair—getting started is often the hardest part.

When you are sure that *Switch* works, make certain that your new tutorial disk is in drive A: (the left-hand drive), type:

SAVE "SWITCH" and press Enter

and wait for the Ok prompt. The program is now permanently saved on your disk. Whenever you want to do graphics from now on, put your tutorial disk in drive A:, start BASIC, and then start the session by typing:

RUN "SWITCH" and press Enter

This will put you on the graphics screen. Try this now. Once you are on the graphics screen, you're ready to roll.

Note: With two-screen systems, it's best to type **SYSTEM** to exit BASIC only while on the monochrome screen. To avoid problems, always run *Switch* from the Color/Graphics Adapter to get back to the monochrome screen before returning to DOS.

A SIMPLE TEST

It's time for a simple test, just to reassure you that you're in the right place. First, type:

NEW and press Enter

This clears the memory of any programs you've run previously and should be done before typing any new program. Next, type Listing 2-2 exactly as shown. (If your screen displays only 40 columns of text, some lines will wrap—just keep on typing until the line is completed.) Press the Enter key after typing each line. Type the command:

LIST and press Enter

to see what you've typed. Any mistakes can be corrected by retyping the incorrect line. If you really get stuck, type **NEW** and start over.

Run the program by typing the command:

RUN and press Enter

The results should be pretty much as shown in Fig. 3 (see the color photograph section) with perhaps some variation due to your display. If the program doesn't work, use **LIST** to check for errors and try again until it does. Once you do get the program running, you are ready to begin acquiring and applying specific knowledge about the PC.

From now on, each listing that you type will be designed to illustrate points made in the text. Try to relate these sample programs to the lessons you are learning. The whole point of this tutorial is to help you understand how to take the tools BASIC provides and apply them to

Listing 2-2. Graphics Display Test.

```

100 REM Program to test whether the display is
110 REM   on the graphics screen.
120 CLS                                'Clear the screen
130 KEY OFF                            'Keys off
140 DEF SEG=0                          'Prepare to check the current screen
150 IF (PEEK(1040) AND 48)<>48 GOTO 200    'If not
160 LOCATE 10,27:PRINT "YOU ARE ON THE WRONG SCREEN!" ' say so
170 LOCATE 25,30:PRINT "PRESS ANY KEY TO EXIT"; ' and make
180 A$=INKEY$:IF A$="" THEN 180 ELSE CLS    ' a proper
190 END                                  ' exit
200 SCREEN 0,1                          'Set COLOR TEXT MODE
210 WIDTH 40                            'Set to 40 characters/line
220 LOCATE ,,0                          'Turn cursor off
230 FOR I=1 TO 15                        'Display 15 lines
240   LOCATE I+5,I+5                    ' starting at 15 locations
250   COLOR I,0,0                      ' in 15 different colors
260   PRINT "It works!!!"
270 NEXT I
280 COLOR 7,0,0                          'Set screen to white and black
290 LOCATE 25,10:PRINT "PRESS ANY KEY TO EXIT"; 'Make a proper
300 A$=INKEY$:IF A$="" THEN 300 ELSE CLS    ' exit
310 END                                  'Return to control BASIC

```

designing useful programs. With this in mind, we will now cover some fundamental concepts of BASIC.

CONCEPTS OF BASIC

Discussing the entire BASIC language in detail would require a separate book. We assume that you either have some familiarity with BASIC or you are willing to pick up the language from the tutorial and from the *BASIC* manual as needed. There are, however, some general points that should be made before we proceed.

BASIC provides us with an environment in which we can create programs, save them to disk, load them back, and erase them. BASIC can also execute commands as we enter them. When BASIC gives you the Ok prompt, it is waiting for you to issue instructions. It is important that you understand that there are two modes of operation in BASIC, and thus two general ways for you to respond.

The first mode is called *direct* or *immediate* mode. In

this mode, we simply type commands, and BASIC immediately executes them. For example, start BASIC, type:

`PRINT "DIRECT MODE"` and press Enter

Notice that BASIC executes the line as soon as you type it. Direct mode is useful for simple examples and for performing small or one-time tasks. Direct mode is not so useful for major tasks, because every command has to be retyped each time you make a mistake or want to perform the task again.

Indirect mode, (also called *deferred*, or *program mode*) is used to enter program lines for later execution. Any line that starts with a number is assumed to be typed in indirect mode. The number becomes the line number, and all the lines in the program are stored for later use in order of line numbers. The lines stored in indirect mode constitute a program. For example, type:

`NEW` and press Enter

to clear any previous programs, then type:

```
30 PRINT "THIRD LINE-INDIRECT MODE" and press
Enter
20 PRINT "SECOND LINE-INDIRECT MODE" and press
Enter
10 PRINT "FIRST LINE-INDIRECT MODE" and press
Enter
```

followed by:

`LIST` and press Enter

You can see that the program lines are reordered according to their number. (The **F1** key may be pressed instead of typing `LIST`, although you must still press Enter. Using the soft keys at the left of the keyboard to generate whole strings in this way can be a real time-saver.) At this point, BASIC does nothing other than store the lines. If you typed the program lines in lowercase, you will notice that BASIC has converted all letters outside the quotes to uppercase. The only time it matters whether you use capitals or lowercase is inside quotes where BASIC keeps letters just as you typed them.

To produce results in indirect mode, type:

RUN and press Enter

BASIC executes the program lines starting with the lowest numbered line. (Instead of typing **RUN** and pressing the Enter key, you can just press the **F2** key. Try this now. The **F2** key sends the command **RUN** and the equivalent of the Enter key, all with one keystroke.)

Indirect mode is used for all large tasks, and for tasks that are executed repeatedly. A program can be run any number of times. Better yet, programs can be saved to disk. For example, save our current program under the file name *Sample* by typing:

SAVE "SAMPLE" and press Enter

(You can press the **F4** key instead of typing **SAVE**", saving four keystrokes.) Now wipe the program from memory with the command **NEW** and press Enter, then type:

LIST and press Enter

or press the **F1** key to check that the program is indeed gone. Now type:

RUN "SAMPLE" and press Enter

and the program is restored to memory and executed. This permanent storage of programs is the great advantage of indirect mode.

Type the command:

LOAD "SAMPLE" and press Enter

This command may be used to restore the program *Sample* to memory without running it. The **LOAD** command wipes out any preexisting program that is in memory. (The **F3** key can be used to send the **LOAD**" command, saving four keystrokes.)

Programs need not be complete to be saved. When you are typing a large program, you should save the program frequently, so that you won't lose your work if the power goes off, or you experience some similar mishap. Also, everything is lost if you exit BASIC with **SYSTEM** and

then restart it. Only a few seconds are required to save a program, but retyping a whole listing can be quite a chore.

STATEMENTS, FUNCTIONS, AND COMMANDS

There are three types of instructions you can give to BASIC. The first of these is the *statement*. A statement directs BASIC to perform an operation, usually in a program. For example, `100 PRINT "TEST"` is a statement.

The second type of instruction is a *function*. A function returns a value. For example, in `A=SIN(X)`, `SIN(X)` is a function that returns the value of the sine of X.

Lastly, *commands* are often used in immediate mode to assist in making programs and perform housekeeping duties, rather than being used in a program. For example, `LIST` is a command. `SAVE` and `RUN` are also commands. You can see that these are commands that help you make programs, rather than being a part of any program. The important distinction here is between a statement which performs an action and a function which returns a value.

The distinction between a command and a statement is not sharply defined, and we will use both terms to describe instructions to the computer. We will also use the term command in a case where either a statement or a function might be used.

THE CURSOR

You have no doubt noticed that there is a flashing underscore at the screen location where your keystrokes appear on the screen. This is called the *cursor*. Whenever we refer to the cursor or the cursor location, we mean the place on the screen marked by the flashing underscore at which the next character typed will appear.

The cursor will appear as a solid block, rather than a flashing underscore, in graphics mode. Also, the cursor will become a half-height block when you are in insert mode in BASIC.

SOME OTHER TERMS

There are a few other terms that we will use frequently which may be unfamiliar to you. One of these is *pixel*. Pixel is derived from *picture element*, and refers to a single dot on the screen, the smallest area that can be controlled by the programmer. A related term is *resolution*. Resolution is the clarity of display possible and is expressed in terms of how many rows and columns of dots can be seen. For example, the highest resolution available on the PC is 200 rows by 640 columns of dots.

Intensity describes the degree of brightness of a color. Each of the colors available on the PC has a brighter version. The brighter colors are called *high-intensity* colors. Thus, on the PC there are two levels of intensity, producing normal and high-intensity colors.

EDITING

Typing programs, particularly long ones, is no easy task. Errors inevitably occur, and changes usually have to be made. BASIC on the IBM PC saves us much trouble when typing programs by providing us with a full-screen editor. This means that we can use the cursor keys (the four arrows on the keypad at the right of the keyboard, shown in Fig. 2-1) to move around the screen and make changes. For example, type:

PRINT "ABCD" and press Enter

Now use the **up-arrow** key (number 8 on the numeric keypad) to go back up to the line you just typed. Use the **right-arrow** key (number 6) to move to the character A. Now type EFGH over ABCD, and press Enter. The modified line will be executed.

You can readily correct lines with these edit keys. The Home, End, Ins, Del, and Backspace keys also provide useful functions. One useful function is that when **Crtl-Home** (the Ctrl and Home keys at the same time) is pressed, the screen is cleared. We are not going to go into detail here on the edit keys, as the *BASIC* manual covers the editing

capabilities of BASIC. We do wish to make one important point, however. When you modify program lines, you must press the Enter key on each line that is modified, or the changes will not register. For example, type:

```
10 PRINT "AAAA" and press Enter
20 PRINT "BBBB" and press Enter
30 PRINT "CCCC" and press Enter
```

LIST the program. Now use the up-arrow key (number 2) to move to line 10. Use the right-arrow key to change AAAA to ZZZZ. Use the **down-arrow** key to move to line 20, and change BBBB to YYYY. Now press the Enter key on line 20. Use the down-arrow key to move below line 30, press the Esc key, and type:

```
LIST and press Enter
```

Note that line 10 is unchanged, because you didn't press Enter on line 10 after changing it. Line 20, on the other hand, is changed, because you did press Enter after changing it. The point is that when you change or type a line, you must press the Enter key on that line for the line to be recognized by BASIC.

OTHER SPECIAL BASIC KEYS

As mentioned at the beginning of this chapter, there are a number of keys that perform special functions in BASIC. The first set of these keys are the ten function keys at the left of the keyboard, which we have already used. These are known as *soft* keys; when pressed, these keys produce a whole string of characters. The soft key assignments are displayed at the bottom of the screen. For example, the **F1** key, when pressed, causes LIST to appear at the cursor location. For our purposes, other useful soft keys include **F2**, **F3**, and **F4**, which produce RUN (and Enter), LOAD'', and SAVE'', respectively. The soft keys can save time, keystrokes, and typing errors, and can even be redefined to meet your needs. One note: a left arrow in the soft key listing at the bottom of the screen, such as that which follows RUN, indicates that the equivalent of the Enter key is sent at the end of the string.

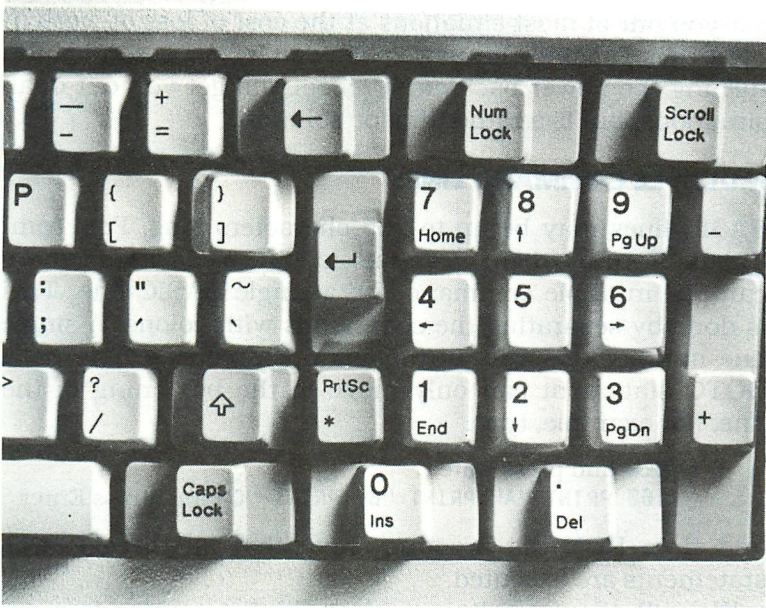


Fig. 2-1. Numeric keypad.

Pressing the **Alt** key in conjunction with a letter will produce a string of characters, much as with the soft keys. For example, press **Alt-R** (hold the **Alt** key down and press the **R** key) and see that the characters **RUN** appear. The **BASIC** manual has a list of all the **Alt** key assignments.

The **Esc** key clears the line the cursor is on and moves the cursor to the left side of the screen. If there are many characters on the screen, **Esc** can be useful for clearing space to type a command.

Finally, always remember that pressing **Ctrl-Break** (pressing the **Break** key while holding the **Ctrl** key down) will stop any **BASIC** program and return control to you. Of course, the program doesn't finish, but it's good to know there's always this safety escape. For example, if you accidentally type a line like **10 GOTO 10** and pressed **Enter** to cause an endless loop, the only way to regain control would be to press **Ctrl-Break**.

As previously noted, **Ctrl-Alt-Del** (depressing the **Ctrl**, **Alt**, and **Del** keys simultaneously) will reboot the **PC** and

get you out of most situations at the cost of loss of work in progress. Turning the power off, waiting 10 seconds, and turning the power back on to reboot will get you out of all situations, but is a rather drastic solution.

MULTIPLE COMMAND LINES

BASIC lines may be up to 255 characters long. Few commands are anywhere near this length, of course, but we can put multiple commands on a single BASIC line. This is done by separating the commands with colons. A single line number refers to all the commands on the line, so a GOTO statement can only move to the beginning of the line. For example, type:

NEW and press Enter

100 PRINT "A":PRINT "B":PRINT "C" and press Enter

and then RUN the program. Note that all three PRINT statements are executed.

We will often use lines with multiple commands separated by colons. One reason is that these lines allow compact programs, and allow us to organize a program into logical portions easily, with related commands placed on the same line. Also, these lines are executed by the computer somewhat faster than the same commands on separate lines.

Note: When the length of the line exceeds the width of the screen, the line wraps around to the left margin of the row below. This is particularly common when the screen is displaying 40 columns of text. You might think that there are now two program lines; however, if the screen were wide enough to display the whole line, it would display it all on the same row. For example, type:

PRINT "XXX . . . XXX"

where "XXX . . . XXX" is 80 X's and press Enter. Note that the line wraps around, but still executes as a single line.

The PC may break in the middle of a word when it wraps to the next line at the right margin. To avoid confusion, we have adjusted our program listings and examples so that wrapping always occurs between words, but when you type, wrapping will occur at other places than shown by our listings. Don't worry about this—the program will execute the same no matter where the lines wrap.

COMMENTS AND GOOD PROGRAMMING STYLE

We cannot teach you everything about programming in this book. We can, however, alert you to the fundamentals of good programming. Throughout this tutorial, we will point out desirable design procedures and will design all our sample programs along proper programming lines.

The single most important advice which we can give you is to put thorough comments in your programs. This means inserting text descriptions that perform no BASIC function into your program for explanatory purposes. BASIC provides two ways of commenting.

The REM statement indicates that all following characters on that line should be ignored by BASIC. For example, type:

```
NEW and press Enter
```

to clear memory and type:

```
100 REM GREET THE USER and press Enter
200 PRINT "HELLO, USER" and press Enter
```

Then RUN the program. Note that line 100 has no effect. On the other hand, line 100 does help make the functioning of the program clearer. Because REM is a statement, it can only appear at the beginning of the line or after a colon.

A single quote also indicates that all following characters on that line should be ignored by BASIC. For example, use NEW to clear memory, and type:

```
100 PRINT "HELLO, USER" 'GREET THE USER and press
Enter
```

RUN this program. The single-quote apostrophe functions

just like the REM statement except that, because the single-quote is not a statement, it can appear anywhere on the line. Any text following a single quote is ignored by BASIC with one exception. A single quote that appears between double quotes, that is, a single quote in a string, is treated as any other character. To see this, type:

```
PRINT "SINGLE QUOTE ' IN STRING" and press Enter  
PRINT ' "SINGLE QUOTE OUTSIDE STRING" and press  
Enter
```

In the first case, the single quote character in the string is printed as a character. In the second case, the single quote character is not in a string and so is considered to mark the beginning of a comment. All text following the single quote is ignored by BASIC.

Please note that because REM statements and single quotes have no effect on the running of a program, they need not be typed when you enter the example programs. This will lessen the typing requirements of the tutorial considerably.

Other good programming practices include doing extensive design before programming, and designing logically separate sections, each of which can be programmed and tested individually. We will go into this in more detail as we take you through the design of several programs in later chapters.

A KALEIDOSCOPE PROGRAM

Let's run a program that will show you some of what BASIC can do on the IBM PC. This program produces a changing kaleidoscope display and is shown in Listing 2-3. Type it precisely as shown. If you make any mistakes, use the editing capabilities of BASIC discussed above to make corrections. Remember to type NEW to clear any previous programs out of memory before typing the new program.

Enter RUN to execute the program, noting the wide range of colors available on the IBM PC. A typical display is shown in Fig. 4 in the color photograph section. Note that good effects can be produced with only a few program lines thanks to the power of Advanced BASIC.

End the kaleidoscope program by pressing any key.

Listing 2-3. Kaleidoscope.

```

100 REM Kaleidoscope program
110 REM This program is inspired by material appearing in
120 REM The Apple II Reference Manual, January 1978, p. 55
130 KEY OFF:WIDTH 40:SCREEN 0,1:COLOR 7,0,0:LOCATE , ,0:CLS
140 LOCATE 25,11:PRINT "PRESS ANY KEY TO EXIT";
150 FOR L=4 TO 40:FOR I=1 TO 12:FOR J=0 TO 11
160 K=I+J:COLOR (J*4/(I+2)+L) MOD 16,0
170 LOCATE K,I+9:PRINT CHR$(219)
180 LOCATE I,K+9:PRINT CHR$(219)
190 LOCATE 24-I,33-K:PRINT CHR$(219)
200 LOCATE 24-K,33-I:PRINT CHR$(219)
210 LOCATE I,33-K:PRINT CHR$(219)
220 LOCATE 24-K,I+9:PRINT CHR$(219)
230 LOCATE K,33-I:PRINT CHR$(219)
240 LOCATE 24-I,K+9:PRINT CHR$(219)
250 IF INKEY$ <> "" THEN 280
260 NEXT J:NEXT I:NEXT L
270 GOTO 150
280 COLOR 7,0,0:CLS
290 END

```

ERRORS

If you encounter errors when trying to run the program, don't worry. You cannot damage the computer, and you can always stop the program by pressing Ctrl-Break.

When an error occurs, BASIC stops processing the current command, stops the program if one is being run, and waits for your next command. If the error is in a program, BASIC tells you what line the error is on. At this point, you must correct the error, then retype the command or rerun the program depending on whether you are in direct or indirect mode. The action necessary to correct the error depends on the type of error encountered.

There are two types of errors. *Syntax* errors occur when you type a command in a form that BASIC cannot understand. For example, type:

```
PIRNT "TEST" and press Enter
```

(The misspelling is intentional.) BASIC will return the message "Syntax error" indicating that it cannot comprehend the command. In indirect mode, BASIC immediately lists the offending program line for you to correct. Syntax errors are nothing more than a nuisance. Because BASIC catches the error for you, and lets you know what program line the error was on, you need merely correct the error. Note that a syntax error is not detected until BASIC attempts to execute the line.

Run-time errors are worse. These errors occur in the course of running the program. There are two types of run-time errors. First, there are those cases when BASIC is given a value that will not work with the specified statement or function or when a command is typed that cannot be executed at that time. For example, type:

```
PRINT SQR(-1) and press Enter
```

Your screen will show "Illegal function call". In this case, we have asked BASIC to calculate a value that cannot be calculated, the square root of a negative number. Don't worry about the specific error; the point is that BASIC has received a command it understands but cannot properly execute.

When this type of run-time error occurs, BASIC, in general, will be less than informative as to the precise nature of the error, though BASIC specifies the line number on which the error occurred. You must examine your program to determine why BASIC could not execute the command on the line.

The second type of run-time error occurs when BASIC runs your program but the results are incorrect. Here there is no error as far as BASIC is concerned; the problem lies strictly in the program logic. Again, you must examine the program to determine the source of the problem. In this case, BASIC provides no help in pinpointing the location of the problem, because, as far as BASIC is concerned, there is no error. The solution here is to trace the program logic all the way from the beginning to the incorrect results.

A complete list of errors is given in the *BASIC* manual.

There are over 70 errors. However, the two error messages that you will receive most will be "Syntax error" and "Illegal function call." When these errors occur, it is up to you to identify their source.

The process of working the errors out of a program is called *debugging*. Legend has it that this term dates back to the first days of the computer era when a malfunction was traced to an insect caught in a relay. Whether this is true or not, the term has become firmly entrenched in computer lingo, so that today there is a debugging program called *DDT* and another called *RAID*.

ADVANCED BASIC

This tutorial will use Advanced BASIC exclusively. We would like to tell you a little bit about this language.

There are three BASICs on the IBM PC. The first is Cassette BASIC. This version of BASIC has all the common BASIC commands found on other popular microcomputers, as well as some simple graphics and sound capabilities. As the name implies, Cassette BASIC can use only cassette storage, rather than disk drives.

Disk BASIC has the capabilities of Cassette BASIC plus the ability to store programs and data on disks and hard disks.

Advanced BASIC has the power of the other two BASICs as well as sophisticated graphics and sound capabilities. In short, Advanced BASIC is the most powerful BASIC available for the PC, and can run any program written for the other BASICs on the PC. The only disadvantage of Advanced BASIC is that it uses more memory than either Cassette or Disk BASIC. Your computer must have at least 48K of memory to use Advanced BASIC.

How does Advanced BASIC measure up to BASICs on other microcomputers?

We'll give you the bad news first. Advanced BASIC is relatively slow, particularly considering that it is running on a 16-bit processor. Advanced BASIC runs at approximately the same speed as both Applesoft® BASIC running on an Apple® II and Microsoft BASIC running on a CP/M

machine; both of these are 8-bit computers. Advanced BASIC is just not that good for time-critical applications.

Now for the good news. Advanced BASIC spends its time providing us with all sorts of nifty features that other BASICs lack. Advanced BASIC can use more memory, allowing larger programs than most BASICs. The graphics, sound, and full-screen editing capabilities of Advanced BASIC are virtually unparalleled by any other microcomputer BASIC. In fact, the only microcomputer BASICs that rival Advanced BASIC for overall power are versions of this same Advanced BASIC written for other computers.

Incidentally, Advanced BASIC is written by Microsoft and is mostly compatible with the 8-bit Microsoft BASIC, apart from its use of more memory, full-screen editing feature, and expanded graphics and sound capabilities. Programs written in 8-bit Microsoft BASIC for CP/M computers will usually run with few modifications on the IBM PC.

It would certainly be nice if Advanced BASIC ran faster. In fact, it would be a good bet that a faster BASIC will be forthcoming fairly soon. Otherwise, Advanced BASIC is an extremely powerful microcomputer BASIC. It is in a class by itself in having the graphics features around which we center this tutorial.

THE TUTORIAL

It's time to begin our tutorial. Before we do, we would like to remind you that you must do more than merely read the tutorial. You must also have your PC running so you can type the examples and gain hands-on experience. Better yet, learn on your own by going beyond what we suggest you do. When we show sample programs, we suggest improvements you may wish to make. Please do so! This book can provide you with a thorough background and can serve as a reference source, but you will not progress as well as you might if you do not do the work—and make mistakes—on your own.

Part 2

A HANDS-ON GRAPHICS TUTORIAL

CHAPTER 3

INTRODUCTION TO THE TUTORIAL

Now that we have the fundamentals out of the way, we can begin exploring in detail the graphics capabilities of the IBM PC. This is not quite as simple as it sounds, because the PC has two main display modes and several submodes.

The two primary display modes are text mode and graphics mode. Text mode is very similar to the display on the monochrome screen except that characters can be displayed in color. In graphics mode, the true graphics power of the PC becomes apparent, with up to 200 rows by 640 columns of dots displayed in up to 4 colors (although not both at the same time).

We will concentrate primarily on the various graphics modes because these are more powerful and because Advanced BASIC takes particular advantage of them, but we will cover text mode as well. Given a little ingenuity, good results can be achieved with text mode, and text-mode effects have the advantage of usually being transferable to the monochrome screen for systems lacking the Color/Graphics Adapter.

The following 15 chapters will be a tutorial on performing graphics from BASIC. First we will cover graphics in medium-resolution graphics mode, then in high-resolution graphics mode, and finally in text mode. We will cover each graphics-related command in turn, making sure that you are familiar with one command before proceeding to

the next. We will discuss the syntax (that is, the way the command is typed so the computer can understand it) and purpose of each command in detail. We will provide immediate-mode examples, which you can type and observe as you read, then explain the operation of complete programs. We will often suggest how you might set about improving the programs on your own, and we strongly encourage you to do this, as well as to experiment at any point to answer whatever questions you might have.

You must use the PC yourself in order to gain any proficiency at programming. You will make mistakes. Though the types of mistakes may change as you gain experience, they never will vanish.

You will find complete applications programs scattered through the tutorial. Designed to show you how to apply what you have learned, these programs will encourage you to be enthusiastic about what the PC can do. These programs intentionally are not as polished as they might be. Feel free to improve them.

BEFORE WE BEGIN

The IBM BASIC manual is the primary reference for the tutorial. If you are a novice programmer, an introductory book on BASIC programming might also be useful.

When the display is on the graphics screen, it is easy to become confused as to whether the screen is in text or graphics mode. In text mode, the soft keys (the function key assignments displayed on the bottom line) are black letters against a white background, and the cursor is a flashing underscore. In graphics mode, the soft keys are displayed as white characters against a black background, and the cursor is a solid block the size of a character within which characters are shown in reverse video (black on white). In medium-resolution graphics mode, only the first five soft keys are shown at the bottom. All ten soft keys are shown in high-resolution graphics mode. When the PC is first booted, the screen is always in text mode.

A final note before we begin our tutorial: we will cover

commands available only in BASIC version 1.1. While BASIC 2.0 has additional graphics capabilities, these are not required to tap the graphics power of the PC. Also, using BASIC 1.1 will allow both BASIC 1.1 and 2.0 users to get maximum benefit from the programs in this book.

CHAPTER 4

MEDIUM-RESOLUTION GRAPHICS—THE PSET STATEMENT

Medium-resolution graphics mode is the “workhorse” graphics mode on the IBM PC. The screen is displayed as 200 rows by 320 columns of dots (called *pixels*), over each of which the programmer has complete control. With this resolution and four colors available at any one time, no other graphics mode offers as powerful a combination of detail and color. Many of the graphics commands are also powerful in this mode. In short, medium-resolution mode has everything needed for excellent graphics: color, fine detail, powerful commands, and even the ability to put text easily onto the screen alongside drawings.

Medium-resolution mode is an excellent choice for games of all sorts, graphs and charts that require more detail than a mere bar graph, and any sort of colorful artistry or design work.

In this chapter, we will show you how to get into medium-resolution graphics mode, how to select the colors you will use, and how to draw a single pixel (dot) and erase it. Also, we will cover several other useful points, especially how to put text onto the graphics screen and the various ways in which you can specify a given pixel.

SOME USEFUL BACKGROUND

There are a few helpful points to be made before we discuss specific commands. First, *parameters* are values

which appear on a line after a statement or function, and which serve to direct the operation of the command. For example, **X** is the only parameter to the function called **SIGN(X)**, and **I** and **J** are parameters to the statement **PSET(I,J)**. There may be no or one or many parameters to a command. We will indicate valid values for all parameters associated with each command that we discuss.

If present, parameters always appear in the same place in the parameter list associated with a command. To put this another way, BASIC knows which parameters are which by their order in the parameter list. The first parameter to the **LOCATE** statement is always the row, the second is always the column, and so on. However, many parameters are optional; that is, they need only be present if the particular action they control is involved. In order for BASIC to know which value applies to which parameter, the comma associated with the parameter must be supplied as a place-keeper if other parameters are specified to the right of the omitted parameter. For example, the statement: **LINE (X,Y)-(X2,Y2),,B** needs a double comma to perform place-keeping for the optional color parameter which is omitted. If the optional parameter, **B**, was also omitted, however, the command would read: **LINE (X,Y)-(X2,Y2)**. Because there is no parameter used to the right of the color parameter, there is no need to hold its place. In fact, a comma without a parameter to the right will cause a "Missing operand" error. Commas are only used when necessary to ensure that BASIC can tell which parameters are specified.

Second, there are always four colors available in medium-resolution color graphics mode—no more and no less. These colors are numbered 0 through 3. Color 0 is always the background color (the background is the default color for the screen when nothing is displayed). This means that setting the color of a pixel to color number 0 will cause that pixel to become indistinguishable from the background; in effect, the pixel will have been erased. The background color is the "setting" upon which graphics are drawn, but it is otherwise no different from

any other color and may always be “drawn” by referencing color number 0.

Finally, we will cover many commands in the course of the tutorial. If you want more detail than we provide on any command, refer to the discussion of that command in the *BASIC* manual. Likewise, you should refer to the *BASIC* manual if you are unfamiliar with any of nongraphics commands that we use. Don't worry too much about nongraphics commands—it is more important to concentrate on the area of current interest.

SETTING UP MEDIUM-RESOLUTION GRAPHICS

The first step in getting BASIC into medium-resolution graphics mode is, of course, to get to the proper screen. If you have a two-screen system, run the program *Switch*, as described in Chapter 2, so that the display will be on your graphics screen (television or composite or RGB monitor), with the BASIC Ok prompt on the screen.

The SCREEN statement selects the mode of operation for the screen. Type the command:

SCREEN 1,0 and press Enter

The screen will clear; this is normal and occurs whenever the screen mode is switched. You are now in medium-resolution graphics mode, with color on.

The general form of the SCREEN command is **SCREEN mode,burst,activepage,display page**

Only the parameters **mode** and **burst** concern us now; **activepage** and **displaypage** apply only to text mode graphics. A value of 1 for mode selects medium-resolution graphics mode, while 2 selects high-resolution graphics mode, and 0 selects text mode. Burst turns color on and off. In text mode, a zero value for burst turns off color and a nonzero value turns it on, while in medium-resolution mode, a zero value for burst turns on color and a nonzero value turns it off. If you encounter problems with this contradictory state of affairs, refer to your *BASIC* manual when setting the screen. Either mode or burst or both may

be omitted if you do not wish to change their values. (As always, more detail is available in the *BASIC* manual if needed.)

Burst does have one unexpected property: it does not turn off color on RGB monitors. (In fact, if burst is turned off in medium-resolution graphics mode, the PC can produce an unofficial set of colors.) Sometimes, it is useful to turn off burst because composite signals tend to produce random color when a color signal is used. For example, the text on a television in medium-resolution mode has stray colored pixels, even though the text is all white. Burst can be used to remove these random colored dots if a truly black-and-white display is desired. None of this applies to RGB monitors, however, for burst has no effect on RGB displays. Just keep in mind that if burst is turned off, the display can be black-and-white on a television screen and colored on an RGB monitor.

For now all you need to remember about the *SCREEN* statement is that if mode is 1 and burst is 0, the screen will be in medium-resolution color graphics mode. We are now ready to examine just which colors we have available.

Medium-Resolution Color Selection

Every color available on the PC can be displayed in medium-resolution graphics mode, but, unfortunately, only in certain combinations. The background color (that is, the color to which the screen and border default and which graphics are typically set against) can be any of the 16 colors available on the PC, as shown in Table 4-1. The other three colors available at any one time consist of one of two sets. The sets are called *palettes* because, like painters' palettes, each is a group of colors from which the PC artist may choose to work.

Palette number 0 contains green, red, and brown, which are colors 1, 2, and 3, respectively (Table 4-2). (Many monitors will display yellow rather than brown.) These colors look excellent on an RGB monitor (see Fig. 5A in the color photograph section), but, unfortunately, often tend to look rather muddy on a television.

Palette number 1 contains cyan, magenta, and white,

Table 4-1. Background Colors

NUMBER	COLOR*	NUMBER	COLOR*
0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	High-Intensity White

* Any of these colors may be selected for the background in medium-resolution graphics mode.

which are, respectively, colors 1, 2, and 3 (Table 4-2). Cyan is a light blue, and magenta is a light red. These colors aren't as vivid on an RGB monitor as those in the other palette (See Fig. 5B in the color photograph section), and can look somewhat washed out if screen brightness is high. However, the colors in palette 1 look just about as good on a television as on an RGB monitor, and because white appears much sharper on a television than any other color, this palette is the palette of choice for television work. (The same is true of any composite monitor.)

Table 4-2. Palette Colors

COLOR NUMBER	PALETTE 0*	PALETTE 1*
1	Green	Cyan
2	Red	Magenta
3	Brown	White

* Only one palette may be selected at any time.

The choice of colors in medium-resolution mode can be difficult. The authors prefer to use palette 1 when both televisions and RGB monitors might be used, because this will ensure that all displays are at least legible. For this reason, most of the medium-resolution examples in this book use palette 1. Unfortunately, palette 0 is often the most visually striking. One solution, of course, is to pro-

vide the end-user the option of selecting the palette he prefers.

The background and palette colors are selected via the **COLOR** statement. Make sure you are still in medium-resolution color mode. Type:

```
COLOR 0,0 and press Enter
```

This will select palette 0 on a black background, so your text will be colored brown (which may appear yellow). The general form of the **COLOR** statement in medium-resolution mode is: **COLOR background,palette** where **background** is the color that drawings are set against (color 0) and **palette** is either 0 or 1 for the palette number. For the palette value any even number will work as well as 0 and any odd number will work as well as 1, but we will use 0 and 1 for consistency. For example, **COLOR 1,1** will select palette 1 on a blue background. This may be hard to read on your display. Type:

```
COLOR 0,1 and press Enter
```

to return to a black background. You will find that a black (color 0) or gray (color 8) background is best for legibility on non-RGB monitors. Note that in medium-resolution graphics mode, when the background color is switched, all dots of color 0 change color immediately. Similarly, when the palette is changed, all dots of colors 1-3 change color immediately.

Type and run the program shown in Listing 4-1 to see the two palettes available in medium-resolution graphics mode, with the **COLOR** statement used to select each of the palettes in turn. Do not worry about how the circles are drawn; we will cover this in later chapters. The key here is the use of the **COLOR** statement to control the palette used.

There are two general points about our programs we would like to make. First, the lines in **FOR . . . NEXT** loops are indented 2 columns. This is a standard practice to make programs more readily understood, as program lines that are related can be readily discerned.

Second, the program in Listing 4-1 terminates with an END statement. A STOP statement would have served as well, and, in fact, because a program terminates when there are no more lines to be executed, neither END nor STOP statements are required. However, the last line in a program is not necessarily the only ending point, or even the ending point at all, so it is a good practice to use END to make the termination point of a program evident.

Listing 4-1. Palette 0 and 1 Demonstration.

```

100 REM Program to demonstrate palettes 0 and 1
110 REM in medium-resolution graphics mode.
120 SCREEN 1,0:CLS:KEY OFF 'Set screen
130 COLOR 0 'Black background
140 REM Do palette 0 first
150 COLOR ,0 'Select palette 0
160 REM Draw circles in colors 1, 2, and 3
170 CIRCLE(80,100),30,1:PAINT STEP(0,0),1
180 CIRCLE(160,100),30,2:PAINT STEP(0,0),2
190 CIRCLE(240,100),30,3:PAINT STEP(0,0),3
200 REM Label screen
210 LOCATE 22,17:PRINT "PALETTE 0"
220 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
230 A$=INKEY$:IF A$="" THEN 230
240 REM Palette 1
250 COLOR ,1 'Select palette 1
260 LOCATE 22,17:PRINT "PALETTE 1"
270 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
280 A$=INKEY$:IF A$="" THEN 280
290 COLOR 0,1:CLS
300 END

```

The program in Listing 4-2 demonstrates each of the 16 colors available for the background color in medium-resolution mode. The variable I is varied from 0 to 15 and used with the COLOR statement to select each of the corresponding background colors in turn.

A gray (color 8) background has an interesting effect on the foreground colors. The colors 8 through 15 are brighter versions of colors 0 through 7, and are called *high-intensity* colors because of this extra brightness. Thus gray (color 8) is the high-intensity version of black (color 0). When the background is gray, the foreground colors become their high-intensity counterparts (in effect, eight is added to the color number (refer to Table 4-1).

Listing 4-2. 16 Background Colors.

```
100 REM Program to show each of the 16 background colors
110 REM   available in medium-resolution graphics mode.
120 REM Set screen
130 SCREEN 1,0:KEY OFF:CLS
140 REM Select each of colors 0 through 15 in turn
150 FOR I=0 TO 15
160   REM Select new background color
170   COLOR I
180   REM Wait a while
190   FOR J=1 TO 800:NEXT J
200 NEXT I
210 REM Reset screen
220 COLOR 0
230 END
```

Hence palette 0, which consists of normal-intensity green, red, and brown (colors 2, 4, and 6) against a black background, becomes light green, light red, and yellow (colors 10, 12, and 14) against a gray background. Likewise, each of the colors in palette 1 becomes a high-intensity version against a gray background.

For example, type:

`SCREEN 1,0: COLOR 0,1: CLS` and press Enter

to select medium-resolution mode with palette 1 against a black background. The text will appear in white. Now type:

`COLOR 8,1` and press Enter

to set the background to gray. Notice how bright the lettering becomes. It is now in high-intensity white (color 15), because the background is high-intensity black. All the other colors in palette 1 are similarly brightened.

Run the program shown in Listing 4-3 to see the difference between the normal and high-intensity palettes. The key here is that the `COLOR` statement is used to select gray, a high-intensity color, for the background color. Consequently, the foreground colors become high-intensity as well.

These high-intensity colors are considerably more vivid than the normal colors, and help to extend the somewhat limited range of colors available in the two standard pal-

Listing 4-3. Medium-Resolution Graphics Mode Intensity Effects.

```

100 REM Program to demonstrate intensity effects
110 REM   in medium-resolution graphics mode.
120 SCREEN 1,0:CLS:KEY OFF      'Set screen
130 REM Do palette 0 first
140 COLOR 0,0   'Select palette 0-normal intensity
150 REM Draw circles in colors 1, 2, and 3
160 CIRCLE(80,100),30,1:PAINT STEP(0,0),1
170 CIRCLE(160,100),30,2:PAINT STEP(0,0),2
180 CIRCLE(240,100),30,3:PAINT STEP(0,0),3
190 REM Label screen
200 LOCATE 23,17:PRINT "PALETTE 0"
210 GOSUB 360      'Delay for viewing
220 REM Now show high-intensity
230 COLOR 8:LOCATE 22,14:PRINT "HIGH INTENSITY"
240 GOSUB 360      'Delay for viewing
250 REM Palette 1
260 COLOR 0,1   'Select palette 1-normal intensity
270 LOCATE 23,17:PRINT "PALETTE 1"
280 REM Clear old label
290 LOCATE 22,14:PRINT SPC(14)
300 GOSUB 360      'Delay for viewing
310 REM Now show high-intensity
320 COLOR 8:LOCATE 22,14:PRINT "HIGH INTENSITY"
330 GOSUB 360      'Delay for viewing
340 CLS:COLOR 0,1
350 END
360 LOCATE 25,9:PRINT "PRESS ANY KEY TO CONTINUE";
370 A$=INKEY$:IF A$="" THEN 370
380 RETURN

```

ettes. With the gray background, all colors except blue and light blue are available in medium-resolution graphics. Just keep in mind that everything we are going to demonstrate with the normal colors can also be done with the high-intensity colors simply by making the background one of the high-intensity colors.

One note on the gray background: if medium-resolution graphics mode is selected with SCREEN 1, and if no COLOR statement is executed to explicitly set the background color, then the background will be gray (color 8), not black (color 0).

MORE USEFUL BACKGROUND

There are two more background points to be discussed before we proceed to graphics. First, after each graphics

command is executed, there is a point known as the *last point referenced*. This point can serve as the automatic starting point for the next graphics command executed. Sometimes, such as when drawing a circle, it is not clear where this point is. In describing each command, we will tell you what point is the last point referenced. Later we will discuss how to make use of this point.

Second, the coordinate numbering system for graphics may be somewhat confusing to the beginning programmer. Each screen location (pixel) is described by the coordinates x and y . The x coordinate indicates pixels measured horizontally, and the y coordinate indicates pixels measured vertically. For all graphics modes, the upper-left corner of the screen is location $(0,0)$, or row 0, column 0, and the lower-left corner is $(0,199)$. In medium-resolution mode the upper-right corner is $(319,0)$ and the lower-right corner is $(319,199)$.

The vertical, or y , coordinates increase from top to bottom, contrary to graphing convention, so your own graphing programs will have to account for this. Also, it may seem strange that numbering begins at zero and not at one, particularly because you will discover later that in text mode the cursor positions start at one and not zero. This is a quirk of the PC you'll learn to work around.

PSET AND PRESET

We are now ready to do some actual graphics work. We will start with the basics—the ability to plot a single point. This is accomplished with the PSET command.

Make sure you are on the graphics screen, in medium-resolution graphics mode with palette 1 and a black background selected. If you are not, the situation can be corrected by booting the PC, typing BASICA, running the program *Switch* if and only if you have both display adapters, and typing:

```
SCREEN 1,0 and press Enter
```

followed by:

```
COLOR 0,1
```


Your display is now set for medium-resolution color graphics, with palette 1 selected on a black background. (We will not repeat this initial sequence again; you should be able to handle it yourself from now on. If you have problems, refer to Chapters 1 and 2 for more detail.)

First type:

`CLS` and press Enter

to clear the screen. `CLS` is a simple but useful statement, having no purpose other than to erase text from the screen.

You will notice that the bottom, or 25th line of the screen, is not cleared. This line displays the assignments of the function keys unless explicitly turned off. Type:

`KEY OFF` and press Enter

to blank this line. The function keys still have their special assignments, but there is no display. Press F1 to see that `LIST` is displayed if you want to reassure yourself. Now press `Esc` to clear the line and type `CLS` again to clear the screen.

As long as we are in immediate mode—that is, as long as we are simply typing commands rather than typing numbered lines into a program for later running—text will always be present on the screen. This is no problem if we frequently clear the screen with `Ctrl-Home` or `CLS` so that we are always working at the top, and keep our graphics away from this area by only graphing points with a y coordinate greater than 100 (that is, in the lower half of the screen). Alternatively, the up-arrow edit key or the Home key can be pressed to reuse the top area of the screen over and over without erasing the display. In any case, if there is anything else on the line on which you are typing a command, either press the `Esc` key to clear the line first, or press `Ctrl-End` to clear from the cursor location to the end of the line. Otherwise, garbage on the line remaining from previous operations can confuse BASIC.

Now type:

`PSET (100,100),3` and press Enter

and you will see a pixel turn white toward the left-center of the screen. The pixel may appear tinted on televisions, but it is white. That is, in truth, about all there is to PSET.

The PSET command is: **PSET (x,y),color** in which **x** and **y** are the column and row coordinates respectively of the pixel and should be within the screen area as described above. In medium-resolution mode, **color** is a value between 0 and 3; 0 selects the background color, while 1, 2, and 3 select colors from the palette that is currently selected via the COLOR statement.

The coordinates are required parameters (otherwise, how would PSET know where to put the pixel?), but color is optional. If color is omitted, the foreground color (color 3, which is white in palette 1 and brown in palette 0) is assumed. The last point referenced by PSET is, of course, the point referenced by parameters **x** and **y**.

Let's experiment a little more with PSET. Remember to clear the screen or reuse the top portion of the screen so that you don't find yourself typing through the graphics. Type:

```
PSET (180,100),2: PSET (181,100),2 and press Enter
```

and voilà—a red (magenta) dot appears. The 2 that appears after the coordinates selects color number 2, and in palette 1 that is magenta. There are actually two red dots plotted. We PSET two dots because on some displays, single colored dots appear erratically or not at all. Type:

```
PSET (200,100),1: PSET (201,100),1 and press Enter
```

and a cyan dot is produced. Now type:

```
PSET (220,100),0: PSET (221,100),0 and press Enter
```

Nothing appears. Why? Remember that color 0 is the background color, so the dots you just plotted in color 0 were indistinguishable from the dots surrounding them. Type:

```
PSET (200,100),0: PSET (201,100),0 and press Enter
```

and the cyan dots vanish, having been set to the background color.

There are two quirks of the PSET statement. First, if PSET is used to plot a point off the screen, no error message is given and no point is drawn. Even though the point is not on the screen, it is still the last point referenced. (This is true of many other graphics commands as well.) The significance of this will become clearer when we discuss relative addressing. For the moment, remember that BASIC doesn't always consider it an error or let you know when you've plotted a point off the screen.

Second, in medium-resolution mode, points PSET with x values between 320 and 639 appear on the screen, although they should not, and wrap around to the left margin to appear two rows below their proper row. For example, type:

PSET(400,100) and press Enter

The dot appears near the left margin, even though it should be off the screen.

PSET has a sister statement, PRESET. PRESET is remarkably similar to PSET; in fact, the two statements are identical unless the color parameter is omitted. In this case, where PSET will default to the foreground color (color 3 in medium-resolution mode), PRESET will default to the background color, color 0. Of course, because PSET(x,y),0 in no way differs from PRESET(x,y), PRESET is not a critically useful statement, but it can serve to make a distinction between drawing pixels (PSET) and erasing them (PRESET).

For example, type:

PSET (100,100) and press Enter

followed by:

PRESET (100,100) and press Enter

to draw and then erase a pixel. You could just as well have erased the pixel with PSET(100,100),0 as with PRESET(100,100).

Almost any graphics function may be performed with PSET and PRESET. Lines, circles, and other shapes are nothing more than a series of pixels strung together to cre-

ate the effect of the desired form. Certainly it is faster and easier to draw a square with a single command than to position and draw each of the pixels individually. PSET and PRESET are nonetheless straightforward and flexible, and can always be made to meet your needs if time is not a critical factor.

For example, PSET can be used in conjunction with a FOR . . . NEXT loop to form lines. Clear the screen with CLS and type:

```
FOR X=100 TO 200: PSET (X,100),1: NEXT and press  
Enter
```

and a blue line will appear. (The FOR . . . NEXT loop causes the PSET command to be executed repeatedly with X equaling 100, then 101, and so on up to 200.) This is all that is required for horizontal and vertical lines; diagonal lines can be draw by commands along the lines of **FOR X=100 TO 150: Y=X-10: PSET (X,Y),3: NEXT** where the relationship between X and Y can be varied to produce lines of different angles. Other forms such as circles can be produced using PSET in conjunction with appropriate calculation (trigonometric functions such as sine and cosine are available), although statements to be covered later can often produce the same results much faster and with considerably less effort.

To get a better feel for what PSET and PRESET can do on your display, select palette number 0 with the command:

```
COLOR 0,0 and press Enter
```

to clear the screen, and experiment with PSET and PRESET with various colors. Experiment further with palette 1. Note which colors show up well and which don't. This information will be useful whenever you are trying to decide which palette you would prefer to use.

PSET And PRESET—An Example

Now we can put our knowledge of PSET and PRESET to use in a complete program. The program, shown in Listing 4-4, uses PSET and PRESET to move a short line

across the screen. Remember that REM statements and single-quote comments are present for explanatory purposes only, and it is not necessary that you type them in when typing the program.

Listing 4-4. PSET and PRESET.

```

100 REM Program to demonstrate use of PSET & PRESET
110 REM   in medium-resolution graphics mode.
120 SCREEN 1,0           'Set medium-res color mode
130 COLOR 0,1           'Select palette 1 for color
140 CLS:KEY OFF          'Clear the screen
150 FOR X=10 TO 300      'Draw moving line 291 times
160   PSET(X,100),2       'Put lead dot on screen
170   PRESET (X-5,100)    'Remove tail end of line
180 NEXT X
190 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
200 A$=INKEY$:IF A$="" THEN 200 ELSE CLS
210 END

```

TEXT IN THE GRAPHICS SCREEN

Text in the graphics screen is essential for such matters as labeling charts, titling presentations, displaying game scores and instructions, and explanations of all kinds. On many microcomputers, inserting text in the graphics screen is a tedious process, requiring the user to draw the characters as he would draw any object. Thus, in order to insert the letter "H" on the screen, the programmer would have to draw three lines, two vertical and one horizontal, in the correct place on the screen.

Happily, we have none of these problems because the PC draws the characters for us. All we have to do is set the cursor location with the LOCATE statement, and then write the text with the normal PRINT command. In fact, we are only able to type BASIC commands in graphics mode because of the PC's ability to put text on the graphics screen.

Clear the screen (you should still be in medium-resolution mode). Type:

```
LOCATE 20,20: PRINT "CURSOR IS HERE"; and press
Enter
```

and observe where the message appears. Now use the edit keys to change the first command to:

```
LOCATE 22,20: PRINT "NOW ON LINE 22"; and press  
Enter
```

and the operation of LOCATE and PRINT should become clear. LOCATE positions the cursor (the point at which the text will appear), and PRINT puts the text string on the screen starting at the cursor location. The values after PRINT can be any string or numeric variables or constants, although you should be aware of the length of the text so that you do not overrun graphics or PRINT past the edge of the screen. Note that the semicolon at the end of the PRINT statement keeps the cursor from going down a line after the text is displayed. PRINTing without a semicolon on lines 24 and 25 will cause the entire screen to move up one line, a process known as *scrolling*. While scrolling can be useful for specialized forms of animation, it is best to avoid it by always using a semicolon at the end of the PRINT statement in the graphics modes. We will discuss PRINT again when we cover text mode, because PRINT is primarily a text-related statement.

We will also discuss the full form of LOCATE in the section on text mode graphics. For now, be aware that the first parameter to LOCATE is the row number (in the range 1 to 25), and the second parameter is the column number (in the range 1 to 40 in medium-resolution graphics mode). Note that you cannot LOCATE in row 25 unless you have executed a KEY OFF statement. By using LOCATE to position the cursor and PRINT to display the text at the cursor location, text can readily be put on the graphics screen where you want it.

If you had trouble following the discussion of the LOCATE and PSET statements, you might want to review Chapter 3 of the IBM BASIC manual, particularly those sections dealing with variables and constants.

There are several drawbacks to this method of putting text on the graphics screen. First, the text size is fixed. Characters are always 8 pixels by 8 pixels in dimension. (In high-resolution mode, this means that characters are

half the width of those in medium-resolution mode.) Second, the text can appear only in normal rows and columns. Because characters are 8 pixels by 8 pixels in dimension, this means that text can only be printed at positions that are multiples of 8 pixels. Thus, the screen is 200 pixels high in medium-resolution mode, but text can appear starting only at pixels 0, 8, 16, and so on. This maintains compatibility with text mode but makes it difficult to get labels in the right places in graphics mode. Third, unless we get a little tricky (to be discussed later), the characters can only appear in the foreground color, yellow in palette 0 and white in palette 1. Fourth, putting characters in the graphics screen via this method is slow; filling any large portion of the screen with text in graphics mode takes several seconds.

Nevertheless, using PRINT to put text on the graphics screen is so much more convenient than any other approach that it is used in most cases.

An Example of Text in the Graphics Screen

Listing 4-5 shows our PSET example program modified to include text labels. Notice that only one LOCATE and PRINT pair is required to place each label on the screen.

RELATIVE ADDRESSING: USING THE LAST POINT REFERENCED

Earlier, we mentioned the *last point referenced* and promised to explain its use later. That time has come. All the screen addresses we have discussed so far have been coordinates such as (0,0) and (160,100), taken as absolute coordinates with the upper left-hand corner of the screen being (0,0). Any particular coordinate pair will always represent precisely the same pixel every time it comes up. This is known as *absolute addressing*. Any program that uses absolute addressing will draw the same form in the same place on the screen every time.

There is another form of graphics addressing known as *relative addressing*. In relative addressing, the coordinates specified are an offset, or movement, from the last

Listing 4-5. Medium-Resolution Graphics Mode Text.

```

100 REM Program to demonstrate use of text in
110 REM   medium-resolution graphics mode.
120 SCREEN 1,0           'Set medium-res color mode
130 COLOR 0,1           'Select palette 1 for color
140 CLS:KEY OFF          'Clear the screen
150 LOCATE 17,14         'Set location and
160 PRINT "MOVING LINE" '   print title
170 LOCATE 15,1         'Set location and
180 PRINT "START"       '   print one label
190 LOCATE 15,33        'Set location and
200 PRINT "FINISH"      '   print second label
210 FOR X=10 TO 300     'Draw moving line 291 times
220   PSET(X,100),2     'Put lead dot on screen
230   PRESET (X-5,100)  'Remove tail end of line
240 NEXT X
250 LOCATE 24,8:PRINT "PRESS ANY KEY TO CONTINUE";
260 A$=INKEY$:IF A$="" THEN 260 ELSE CLS
270 END

```

point referenced (this is why the last point referenced by each command is important). To specify relative addressing, describe the screen address with **STEP(x,y)** where **x** is the number of pixels to be added horizontally to the last point referenced, and **y** is the number of pixels to be added vertically to the last point; the resulting coordinate is the new screen location. Negative values may also be used to indicate movement up or to the left.

This may sound complicated, but it's really not. Just remember that with relative addressing the reference, or (0,0), point is the last point referenced, rather than the upper-left corner.

For example, in medium-resolution graphics mode, clear the screen and type:

PSET (100,100) and press Enter

which makes the last point referenced (100,100). Now type:

PSET STEP(10,10) and press Enter

which will put a dot 10 pixels below and to the right of the first dot. Next type:

PSET STEP(-30,10) and press Enter

which will put a dot 10 pixels below and 30 to the left of the second dot. Note that **PSET STEP(20,-20)** would return the last point referenced to the initial dot.

Relative addressing can be used with most of the graphics commands we will discuss later, by simply substituting **STEP(x,y)** for the normal absolute coordinate pair. Experiment now with **PSET** and **PRESET** using relative addressing until you've gotten the hang of viewing the screen from a relative perspective. See what happens if you plot off the screen. Such error conditions can be important in anticipating potential problem areas in your programs.

Relative mode can be useful. For example, suppose you want to put a symbol representing a factory at any of several locations on any chart produced. Using absolute addressing, each coordinate for **PSET** would have to be calculated explicitly.

Now, suppose instead that you wrote a section of program code to draw the symbol using relative addressing. To draw the symbol at any screen location, you would execute one **PSET** to set the starting point for the symbol, and then execute the logo-drawing code. No calculation beyond the first point would be required. The relative addressing approach is certainly easier to use in this case, and better any time you need to draw the same form in several locations.

There are several points to note concerning the last point referenced. Even if the last graphics command referenced a point off the screen, the last point referenced is that point. For example: **PSET (160,90): PSET STEP(0,300) PSET STEP(0,-280)** will produce a third dot 20 pixels below the first, even though the second plot was off the screen. Also note that there is no error message returned when plotting the offscreen point. Do not count on **BASIC** to let you know when you've made a mistake!

Finally, if no last point referenced has yet been set, the last point referenced is the center of the screen, (160,100). Each time the screen is cleared, the last point referenced is

reset to this point. This makes sense—if the screen is cleared, the last point referenced no longer has any meaning, as the drawing it was related to no longer exists.

In case you were wondering, there is no BASIC command that lets you find out where on the screen the last point referenced is. While this can certainly be a nuisance, it's another one of those things that you'll just have to live with.

An Example of Screen Addressing Modes

Listing 4-6 shows a program which performs both absolute and relative addressing. Make sure you understand the use of STEP in this program to specify relative screen addresses.

Listing 4-6. Absolute and Relative Screen Addressing.

```

100 REM Program to demonstrate absolute and relative
110 REM   screen addressing.
120 SCREEN 1,0           'Set medium-res color mode
130 COLOR 0,1           'Select palette 1 for color
140 CLS:KEY OFF          'Clear the screen
150 FOR I=1 TO 200       '200 times
160   PSET(50+I,50),1     ' draw moving dots
170   PSET STEP(20,30),2 ' in three
180   PSET STEP(20,30),3 ' colors
190 NEXT I
200 LOCATE 24,8:PRINT "PRESS ANY KEY TO CONTINUE";
210 A$=INKEY$:IF A$="" THEN 210 ELSE CLS
220 END

```

A word of caution about relative addressing: if the last point referenced is accidentally placed off the screen, the overall picture will sometimes, although not always, be garbled. For this reason, we suggest that you make every effort to avoid addressing points off the screen.

CHAPTER 5

THE POINT FUNCTION

With the PSET statement, we have full control over what is put on the screen, but we have no way to tell what is already there. The latter ability would give us full control over the graphics screen; in time, given PSET and a function for checking screen contents, we would be able to perform virtually any graphics application.

Why is the ability to detect screen conditions so important? Imagine that we were designing a game in which, for example, a picture of a cyan whale is moving around the screen under player control, avoiding the white icebergs and attempting to collide with the magenta fish. It would be useful to test whether certain pixels were already drawn—that is, differing from the background color, indicating the presence of an object at that location. Furthermore, the color of the pixel can indicate what object has been hit.

Or suppose that we wished to draw a chart with criss-crossing lines of red and blue. The red lines are more important, so we would want to draw blue lines only at pixels that aren't already red. Again, some way to test a pixel in advance is required.

The POINT function is designed for just such applications. For instance, clear the screen and type:

```
PSET (100,100),3 and press Enter
```

then type:

```
PRINT POINT(100,100) and press Enter
```

The result, 3, indicates that the pixel at (100,100) is white, as it should be, because we just made it white ourselves.

The syntax of the POINT function is **z=POINT(x,y)** with **x** and **y** as the coordinates of the pixel to be tested. We have previously dealt only with statements, but POINT is a function and always returns a value. This value can be used in an IF statement or in a PRINT statement (as in the previous example), or it can be assigned to a variable. This value is, as regards usage, no different than that of a normal variable such as I or J.

The value returned by POINT can then be used to test the color of a pixel. For example, if POINT returns a color of 0, then the pixel is the background color.

In medium-resolution mode, values returned by POINT may be 0, 1, 2, and 3, corresponding to the three colors available in the currently selected palette. If the selected palette is palette number 0 and POINT returns the value 1, the pixel is colored green; were palette 1 to be selected, the same value would represent cyan.

The POINT function does differ markedly from other graphics commands in one respect. Clear the screen and type:

```
PSET (100,100) and press Enter
```

to set the last point referenced. Then type:

```
PRINT POINT STEP(1,1) and press Enter
```

and get the result 0, indicating that this pixel is black. However, **PRINT POINT STEP(0,0)** will return 3, where one might well expect 0 again, because the last POINT referred to a pixel that was black.

Unlike the other graphics-related commands we will discuss, POINT does not affect the last point referenced. Relative addressing can be used with POINT, but the last point referenced always remains unchanged by the operation of the POINT function.

POINT is useful in any case where the screen's contents are of interest. Collision detection is a particularly good application. In "shoot-'em-up" games, an enormous number of time-consuming collision checks usually done by

comparing coordinates can be avoided by simply checking for collisions in the screen via POINT.

AN EXAMPLE OF THE POINT FUNCTION

POINT is of no particular use in and of itself. Pixels must have previously been drawn with some purpose in order for the values returned by POINT to have any meaning. Our example of the POINT function involves the PSET and PRESET statements as well, and contains some of the elements we will use in later chapters when we design games.

A program to demonstrate the use of the POINT statement is shown in Listing 5-1. Type and run the program, then examine the listing to connect the action on the screen to the BASIC program. The key to this program is that on lines 330 and 370, the POINT function is used to test whether the new location of the dot is color 0. If it is not color 0, then it must be one of the walls of the box and so it must be time for the moving ball in the program to bounce.

Lines 110-130 set up the medium resolution screen. Lines 140-230 draw the box and the two lines in the middle of it. Lines 240-280 set the initial location and motion of the ball. The FOR . . . NEXT loop starting at line 290 and ending with line 420 moves the ball.

Within the ball movement loop, lines 300 and 310 set aside the old ball location, so that on line 400 the computer knows where to erase the ball. Lines 320-350 move the ball in the x direction, check for collision with a wall, and, if a wall is encountered, reverse direction and move the ball in the new direction. Likewise, lines 360-390 handle the movement of the ball in the y direction. Line 400 erases the ball at the old location, and line 410 draws the ball at the new location.

We are not going to explain this in more detail here, because the main object of this program is to demonstrate the use of the POINT function, and because we will discuss a similar program at great length in Chapter 11. Do, however, look over the program, because this approach to

moving an object is useful and frequently encountered. Also, make certain that you understand the role that the POINT function plays in determining whether the ball has struck something on the screen.

Listing 5-1. POINT Function.

```

100 REM Program to demonstrate the POINT function.
110 SCREEN 1,0 'Set med-res color mode
120 COLOR 0,1 'Select palette 1 for color
130 KEY OFF:CLS 'Clear the screen
140 FOR I=50 TO 100 'Sides of box are 50 long
150 PSET (110,I) 'Left side
160 PSET (210,I) 'Right side
170 NEXT I
180 FOR I=110 TO 210 'Top and bottom are 100 long
190 PSET (I,50) 'Top side
200 PSET (I,100) 'Bottom side
210 NEXT I
220 LINE (130,75)-(150,75) 'First line in middle
230 LINE (170,75)-(190,75) 'Second line in middle
240 DOTX=130 'Initial x coordinates
250 DOTY=80 'Initial y coordinate
260 DOTXINC=1 'Initial x amount to move
270 DOTYINC=1 'Initial y amount to move
280 PSET (DOTX,DOTY) 'Put initial dot on
290 FOR I=1 TO 500 'Move dot 500 times
300 DOTXOLD=DOTX 'Save old x for use when erasing
310 DOTYOLD=DOTY 'Save old y for use when erasing
320 DOTX=DOTX+DOTXINC 'Move in x direction
330 IF POINT(DOTX,DOTY)=0 THEN 360 'See if hit wall
340 DOTXINC=-DOTXINC 'Hit wall, so reverse x motion
350 DOTX=DOTX+2*DOTXINC ' & move in new x direction
360 DOTY=DOTY+DOTYINC 'Move in y direction
370 IF POINT(DOTX,DOTY)=0 THEN 400 'See if hit wall
380 DOTYINC=-DOTYINC 'Hit wall, so reverse y motion
390 DOTY=DOTY+2*DOTYINC ' & move in new y direction
400 PRESET (DOTXOLD,DOTYOLD) 'Erase old dot
410 PSET (DOTX,DOTY) 'Draw dot in new location
420 NEXT I
430 PRESET (DOTX,DOTY) 'Erase last dot
440 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
450 A$=INKEY$:IF A$="" THEN 450 ELSE CLS
460 END

```


CHAPTER 6

THE LINE STATEMENT

We now proceed to the **LINE** statement, the first of the higher-level graphics commands. By “higher-level” we mean commands which perform tasks that would take many executions of **PSET**. Higher-level graphics commands save not only trouble but also a great deal of time, because they run much faster than the equivalent collection of **PSET**s.

The **LINE** statement allows us to draw lines at any angle and to draw rectangles whose edges are parallel to the sides of the screen. **LINE** also allows us to fill in the rectangle we draw with any color.

To get an idea of what **LINE** can do, enter medium-resolution graphics mode with palette 1 selected on a black background (from now on we will assume you are using palette 1 unless otherwise noted). Clear the screen, type:

```
LINE (100,100)-(100,200) and press Enter
```

Use the up-arrow key to return to the top line, press **Esc** to clear the line, and type:

```
LINE (150,100)-(200,150),2 and press Enter
```

Similarly, type:

```
LINE (220,100)-(240,120),,B and press Enter
```

and:

```
LINE (260,100)-(290,130),1,BF and press Enter
```

The general form of the **LINE** statement is **LINE (x1,y1) -**

(**x2,y2**), **color**, **BF**. **x1** and **y1** are the coordinates of the start point of the line, and **x2** and **y2** are the coordinates of the end point. In medium-resolution mode, **color** may be 0 for the background color, or any of colors 1, 2, or 3 from the active palette.

The parameter **B**, if present, indicates that a box should be drawn, with (**x1,y1**) as one corner of the rectangle, and (**x2,y2**) as the corner at the other end of the diagonal. The parameter **BF** causes a box to be drawn and filled with the color specified by the color parameter. All parameters to the **LINE** statement are optional except for **x2** and **y2**. The last point referenced by the **LINE** statement is the point (**x2,y2**).

LINE is the first really complex command we've encountered, so we'll cover it one part at a time, with "hands-on" examples.

The (**x1,y1**) and (**x2,y2**) parameters are the start and end points of the line to be drawn. Clear the screen by pressing Ctrl-Home, and type:

```
LINE (100,50)-(200,150) and press Enter
```

A diagonal line will appear; this is **LINE** in its simplest use.

Either coordinate may be specified in either relative or absolute form. (The above example uses absolute screen addressing.) Type:

```
PSET (90,60)
```

to set the last point referenced, then type:

```
LINE STEP(10,20)-(200,180) and press Enter
```

which will draw a line parallel to the first. If relative mode is used for the second point, the first point is used as the last referenced point, and the second point is calculated in relation to the first point. For example, **LINE (100,20) - STEP (100,100)** will draw another parallel line. Relative mode may be used for both coordinates; type:

```
PSET (0,0): LINE
```

```
STEP(100,60)-STEP(100,100) and press Enter
```

to create a fourth line.

The starting point may be omitted. In this case, the last point referenced is used as the starting point, just as if `STEP(0,0)` had been specified for the first coordinate. Clear the screen and type:

`PSET (100,100): LINE-(190,190)` and press Enter

noting that the starting point of the line is, indeed, at location `(100,100)`.

Lines can be drawn in any of the four available colors by varying the color. For example, `LINE (100,100) - (200,100), 2` draws a magenta line.

The last parameter is either the letter B or the letters BF. These are not numeric variables like x or y, but rather alphabetic characters. Only one of these two parameters can be specified, because their operations are exclusive of one another. If the B parameter is present, it directs the drawing of a box or rectangle. For an example of the operation of the B parameter, clear the screen and type:

`LINE (100,50)-(200,150),3,B` and press Enter

This will draw a box with corners `(100,50)`, `(200,50)`, `(200,150)`, and `(100,150)`. Were it not for the B option, four LINE statements would be required to draw the same box. Worse yet, four FOR . . . NEXT loops with PSET statements would be required to produce the same result.

The sides of the box are always parallel to the edges of the screen. There is, alas, no way to specify a box tilted at an angle, although one could combine the lines produced with four separate LINE statements to produce such a box. The starting coordinate pair `(x1,y1)` is one corner of the box, and the second coordinate pair `(x2,y2)` is the opposite corner. The four corners of the box are `(x1,y1)`, `(x1,y2)`, `(x2,y1)`, and `(x2,y2)`.

The optional BF parameter indicates that a box should be drawn and filled with the selected color. For example, `CLS: LINE (100,50)-(200,150),2,BF` draws a magenta box.

In general, out-of-range coordinates to LINE are changed to the nearest valid (on-screen) value and no error message is given. Any y values greater than or equal to 200 are

treated as 199, any negative x or y values are treated as 0, and x values greater than 319 are treated as 319.

And that's really all there is to the LINE statement. While LINE is certainly more complicated than PSET, the advantages in speed, clarity, ease of use, and compactness of code are great. The LINE statement is not the most versatile of the graphics commands, but there is no faster way to draw an object of solid color available in IBM BASIC.

The LINE statement is useful for drawing bar graphs, line figures, and graphing and plotting axes. Continuous graphs, such as that of sales over time, are made by omitting the starting-point coordinate pair for all but the first point plotted, so that a line is drawn from each point to the next, producing an unbroken plot.

AN EXAMPLE OF THE LINE STATEMENT

The program shown in Listing 6-1 uses the LINE statement to create the image of a face. Because we are restricted to using lines, the image is angular, but notice how quickly the image is drawn, including the solidly colored areas of the eyes. We initially drew the eyes solidly colored, and then drew a smaller, solid box of color 0 to restore the pupil areas to the background color. The operation takes place so fast that it is impossible to see the solidly colored place eyes for the moment they are on the screen before the pupils are drawn. We also omitted the starting coordinate for LINE when drawing the last two segments of the mouth; this is a handy way to produce smoothly connected lines.

Listing 6-1. LINE Statement.

```

100 REM Program that demonstrates the capabilities
110 REM   of the LINE statement.
120 SCREEN 1,0           'Set medium-res color mode
130 COLOR 0,1           'Select palette 1 for color
140 KEY OFF:CLS          'Clear the screen
150 LINE (110,50)-(210,150),2,B 'Head
160 LINE (130,70)-(150,90),1,BF 'Left eye
170 LINE (135,75)-(145,85),0,BF 'Left pupil
180 LINE (170,70)-(190,90),1,BF 'Right eye
190 LINE (175,75)-(185,85),0,BF 'Right pupil
200 LINE (125,110)-(140,125),3 'Left side of mouth
210 LINE -STEP(40,0)          'Center of mouth
220 LINE -STEP(15,-15)        'Right side of mouth
230 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
240 A$=INKEY$:IF A$="" THEN 240 ELSE CLS
250 END

```

CHAPTER 7

ELLIPSES, ARCS, AND WEDGES—THE CIRCLE STATEMENT

So far, we've learned how to control a single pixel, and we've learned how the LINE statement can save us a FOR . . . NEXT loop and considerable time when drawing lines. Drawing most objects, however, requires more than simple straight lines; unfortunately, nonlinear forms are usually much more difficult to draw, often requiring complex calculations and considerable code. On most microcomputers, forms such as circles and arcs are drawn with machine-language subroutines which are tedious to use and difficult to write and debug.

Once again, the PC simplifies programming. The CIRCLE statement gives us the ability to draw circles, arcs, and wedges of any size and radius with a single statement. With the CIRCLE statement, the PC graphics programmer is free to concentrate on the conceptual aspects of many graphics applications rather than on esoteric programming techniques.

Make sure that you are in medium-resolution mode with palette 1 selected. Clear the screen and type:

```
CIRCLE (40,120),30 and press Enter
```

followed by:

```
CIRCLE (140,120),50,3,0,2 and press Enter
```


and:

CIRCLE (240,120),40,3,-1,-5,2 and press Enter

Notice the variety of forms that are easily produced with the **CIRCLE** statement.

There can be as many as seven parameters to the **CIRCLE** statement, but we will start off with a simpler form of the statement and build up to the full specification. In the basic form, **CIRCLE (x,y),radius,color**, **x** and **y** are the coordinates of the center of the circle (in either absolute or relative), **radius** is the distance from the center to the outer edge of the circle, and **color** selects the color of the edge of the circle, with color 0 the background and the other three colors belonging to the selected palette. The color parameter is optional and defaults to color 3. Note that radius is the distance from the center of the circle to the edge as measured in pixels on the screen. The importance of this will become apparent shortly. The last point referenced by the **CIRCLE** statement is the center of the circle.

To draw a magenta circle, for example, type:

CIRCLE (160,100),50,2 and press Enter

To draw another circle concentric to the first, type:

CIRCLE (160,100),30,1 and press Enter

and:

CIRCLE (160,100),40 and press Enter

to produce yet another.

CIRCLE has an error-handling characteristic worth noting. If any part of the circle is off the screen, that portion is simply not drawn; no error message is given. The last point referenced is the center of the circle, even if that point is off the screen. Entering **CIRCLE (100,100), 30: CIRCLE STEP (0,1000), 30** followed by **CIRCLE STEP (0,1000), 30: CIRCLE STEP (0,-2000), 50** produces two concentric circles. This result is straightforward; just remember that off-screen circles are handled like any other circles except the off-screen portions aren't shown.

And remember that there is no error message produced, so that it will not necessarily be obvious immediately if you have erroneously plotted a circle off the screen.

Listing 7-1 draws circles of various sizes and colors (see Fig. 6 in the color photograph section). The function **R MOD 4** on line 150 returns the remainder of the division of R by 4, so that values can only be 0 through 3. These values correspond to the four colors available in medium-resolution mode. As the value of R varies from 1 to 90 in increments of 5, the color parameter will cycle through each of the four colors.

Listing 7-1. Circle Statement.

```

100 REM Program to demonstrate the CIRCLE statement.
110 SCREEN 1,0:COLOR 0,1      'Set screen
120 KEY OFF:CLS                'Clear screen
130 REM Draw circles of various sizes and colors
140 FOR R=1 TO 90 STEP 5      'Step through 18 radiuses
150   CIRCLE(160,100),R,R MOD 4 'Color based on radius
160 NEXT R
170 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
180 IF INKEY$="" GOTO 180 ELSE CLS
190 END

```

There is a useful trick to be learned from Listing 7-1. If a key has been pressed, the **INKEY\$** function returns that character; if not, **INKEY\$** returns a null string. **INKEY\$** can be used to detect keystrokes with a program line, such as **IF INKEY\$ <> "" THEN** instructions to handle key pressed.

On line 180, **INKEY\$** is used to keep BASIC from printing anything on the screen, and thus "messing up" the display, until a key is pressed. A similar line can be inserted after any display has been drawn, giving you time to admire your handiwork. This is a useful trick and you may have noticed it in previous programs.

ARCS AND WEDGES

A circle can be used to represent the sun, a target, or a ball. A collection of circle fragments, or arcs, can look, on a computer screen, like hills, waves, seagulls, or plots on a diagram. Similarly, a circle with a wedge removed, can

appear to be a pie, a pie chart, or even the most popular arcade creature of all time. When we add two parameters to the CIRCLE statement, we make it possible to produce these arcs and wedges.

The expanded CIRCLE statement is **CIRCLE (x,y), radius, color, start, end**. **Start** and **end** are the two ends of the arc to be drawn. The values of start and end are specified in radians. When measuring in radians, for example, the three-o'clock point on the circle is 0, and the values increase counterclockwise to 3.14 (pi) at nine o'clock, and to about 6.28 (2 times pi) at the three o'clock position. Note that the values for start and end must be between -6.28 and 6.28, which is contrary to standard practice. The meaning of negative values for start and end will be explained subsequently.

The arc is drawn beginning at start and continuing counterclockwise to the end value. This is true even if the arc wraps around past the three o'clock, or zero radians, point.

Start and end are optional parameters. Start defaults to 0 and end defaults to 6.28. This means that the defaults start at the three o'clock position and end, after rotating counterclockwise through the entire circle, back at the three o'clock position. If both parameters are omitted, the whole circle will be drawn. Remember that if you do use the start and end parameters, you must either use the color parameter or insert a double comma to hold the color parameter's place.

Let's draw a few arcs to get used to using the start and end parameters. Clear the screen and type:

```
CIRCLE (160,100),30,,0,3.14 and press Enter
```

to draw the upper half of a circle. Note the use of two commas to hold the place of the omitted color parameter. **CIRCLE (160,100),30,2,3.14,0** draws the other half in color number 2. Clear the screen and type:

```
FOR I=10 TO 290 STEP 20: CIRCLE  
(I,100),10,2,3.14,0: NEXT and press Enter
```

to draw a series of peaks. Remember that arcs are always

drawn counterclockwise, so a start and end of 3.14 and 0, respectively, specify the drawing of the bottom half of a circle.

We can draw more than circle halves. Clear the screen and type:

```
FOR I=10 to 50 STEP 10: CIRCLE  
(160,100),I,,2.3,4.0: NEXT and press Enter
```

to draw smaller arcs.

Clear the screen and type:

```
CIRCLE (160,100),40,,4,3 and press Enter
```

to see how arcs wrap past the three o'clock point if the end parameter is less than the start parameter.

A wedge, or pie slice, is nothing more than an arc with lines connecting the end points to the center of the circle. With the CIRCLE statement, if either start or end is a negative number, that point is connected to the center of the circle. (The start or end value is then used as if it were a positive number; this is not the same as adding 2 times pi to the number, which is standard mathematical practice.) If both points are negative, a wedge is formed. For example, clear the screen and type:

```
CIRCLE (160,100),30,, -1.57, -3.14 and press Enter
```

to draw a pie quarter, and type:

```
CIRCLE (240,100),30,2, -3.14, -1.57 and press Enter
```

to draw the pie the quarter was cut from. Only one of the sides of the wedge need be drawn, as evidenced by typing:

```
CIRCLE (100,100),20,, -1.57,4 and press Enter
```

and:

```
CIRCLE (100,160),30,,1.57,-4 and press Enter
```

These wedge-drawing capabilities of the CIRCLE statement will be useful later when we create a pie-chart drawing program.

ASPECT AND THE SCREEN

Another capability of the CIRCLE statement is the drawing of ellipses. For our purposes, an ellipse is a circle that

is not necessarily perfectly round (an egg in profile, for example), although that is not quite the technical definition. Ellipses and sections of ellipses are frequently encountered in graphics work—in fact, you will discover the circles we have already plotted were, properly speaking, noncircular ellipses.

Admittedly, the circles looked perfectly circular, but the separation on the screen between two pixels is greater in the vertical direction than in the horizontal direction. The two values are close enough so the effect of this is unnoticeable for a small drawing. For example, draw a square with:

`CLS: LINE (50,50)-(150,150),,B` and press Enter

The difference between the horizontal and vertical dimensions should be apparent if you look closely. Leave the image on the screen and draw a circle of the same size by typing:

`CIRCLE (100,100),50` and press Enter

The effect will now be unmistakable.

Just how irregular the standard screen is is hard to say. The BASIC manual has a confusing discussion of screen aspect, in which the values $4/3$, $8/5$, and $24/20$ are mentioned. (Different versions of BASIC have different discussions, but none of the manuals are particularly clear.) On a standard screen, 5 pixels vertically cover the same distance as 6 horizontally, and so $5/6$ is the proper correction for the aspect of the screen.

The CIRCLE statement automatically compensates for the screen's irregularity by drawing ellipses elongated to compensate; the effect is a figure that looks perfectly round. Measured in inches, the figures we have produced with CIRCLE are indeed circles, but, measured in pixels, they are not. This is why we need to remember that radius is measured in pixels rather than in inches. In BASIC, the radius is not the distance from the center to the edge of the circle at all points, but rather only at two special points. This brings us to the concept of *aspect ratio*.

The dimensions of an ellipse can be measured both hori-

zonally (on the x axis) and vertically (on the y axis). The ratio of the y axis to the x axis is the aspect ratio, which, for a true circle, is 1. For consistency, all aspect ratios from now on will be discussed in terms of length in pixels rather than inches. For the circles we plotted, the aspect ratio was 5/6; the slight shortening of the y axis produces the visual effect of a circle. A shape like a flying saucer would have a low aspect ratio, perhaps 1/5, while a shape like an egg standing on end would have a high aspect ratio of perhaps 4/1.

The shape of an ellipse drawn with **CIRCLE** is defined by the aspect ratio. The complete form of the **CIRCLE** statement is **CIRCLE (x,y),radius,color,start,end,aspect** where **aspect** is the aspect ratio of the circle. For example, type:

```
CIRCLE (160,100),50,,,,1: CIRCLE (160,100),70
and press Enter
```

and the noncircularity of a circle with an aspect ratio of 1 is readily apparent.

With aspect ratio in mind, we can now consider the actual meaning of the **radius** parameter, which alone determines the size of the ellipse. Simply put, radius is measured in pixels along the longer of the two ellipse axes. If the aspect ratio is 1, for example, then radius is the distance from the center to the edge of the circle along both axes. If the aspect ratio is less than 1 (this means that the x axis is the longer axis) then radius is the distance from the center to the edge of the circle along the x axis only. Similarly, if the aspect ratio is greater than 1, radius is the distance from the center to the edge along the y axis only. This means that the radius of the default circle (the circle produced by the **CIRCLE** statement in the absence of the aspect parameter) is measured along the x axis only, because aspect defaults to 5/6. This can be useful knowledge when doing precision graphics.

To see the aspect parameter in action, type and run the program in Listing 7-2. This program first decreases the aspect ratio from 10 to 1 while holding radius constant; notice that the length of the y axis is constant from circle

to circle, while the x axis steadily lengthens. This is because the aspect ratio is greater than 1, so radius is measured in pixels along the y axis. The aspect ratio is then decreased from 1 to 1/10; here, the length of the x axis remains constant at radius, while the y axis steadily shortens.

Listing 7-2. Varying Aspect Ratio with CIRCLE Statement.

```

100 REM Example of varying the aspect ratio with
110 REM   the CIRCLE statement.
120 REM Initialize screen
130 KEY OFF:CLS:SCREEN 1,0:COLOR 0,1
140 REM Draw ellipses of different widths
150 FOR ASPECT=10 TO 1 STEP -.1 'Use 10 aspect ratios
160   GOSUB 260 'Draw ellipse
170 NEXT ASPECT
180 REM Draw ellipses of different heights
190 FOR ASPECT=1 TO .1 STEP -.1 'Use 10 aspect ratios
200   GOSUB 260 'Draw ellipse
210 NEXT ASPECT
220 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
230 A$=INKEY$:IF A$="" THEN 230 ELSE CLS
240 END
250 REM Subroutine to draw new ellipse
260 CIRCLE (160,100),60,3,,,ASPECT 'Draw new ellipse
270 REM Wait a bit so the new ellipse can be viewed
280 FOR PAUSE=1 TO 500:NEXT PAUSE
290 RETURN

```

In this example, the subroutine starting at line 250 draws the new ellipse. Note that there is a delay so the viewer has time to appreciate one circle before the next is drawn. The FOR . . . NEXT loop at lines 150-170 runs through aspects from 10 to 1, while the FOR . . . NEXT loop at lines 190-210 draws circles with aspects from 1 to 0.1, stepping by 1/10.

Only a few lines are required to draw a wide range of curved shapes with the complete CIRCLE statement, and with sufficient ingenuity, arcs of various aspects and sizes can be pieced together to produce virtually any curvature.

As a final note on ellipses, the BASIC manual states that an aspect ratio of 1 may produce circles (which appear to be slightly ellipsoid) that are more attractive than the

default 5/6 aspect ratio. This is because circles are drawn symmetrically, with fewer jagged edges, when the aspect ratio is 1. The manual states that circles with an aspect ratio of 1 are also drawn somewhat faster; in actual practice, the increase in drawing speed seems negligible, particularly given the rather slow speed of the CIRCLE statement. Lastly, the BASIC manual indicates that aspect is the ratio of the x axis to the y axis, when, in fact, aspect is the ratio of the y axis to the x axis.

We have touched on some of the applications for the CIRCLE statement, and we will discuss other applications in later example programs. You may have noted the slow execution speed of the CIRCLE statement and thought that this limits the usefulness of CIRCLE when rapid drawing must be performed, but this is not so. The graphics GET and PUT statements, to be covered later, can work with the CIRCLE, PAINT, and DRAW commands to generate graphics displays with remarkable rapidity. With the help of GET and PUT, even arcade-type games of many kinds can be written in PC BASIC without the help of machine-language routines, and that is a claim that few BASICs can make.

The CIRCLE statement is a key element of the *Pie Chart* program discussed in Chapter 9.

CHAPTER 8

THE PAINT STATEMENT—THE ARTIST'S BRUSH

Thus far, we have acquired considerable control over drawing lines of various sizes, shapes, and colors on the screen. We have not, however, developed much of an ability to program the PC to fill in these areas (the BF option to the LINE statement is the exception). Because the color set of the PC includes many light and pastel colors, large areas of solid color often produce vivid effects. What we need is a “brush” capable of coloring our computer-generated bars on histograms, wedges in pie charts, and countries on maps.

We get all of the above in a simple, yet remarkably versatile, form in the PAINT statement. Clear the screen in medium-resolution graphics mode and type:

```
CIRCLE (160,100),40,2 and press Enter
```

followed by:

```
PAINT STEP(0,0),2 and press Enter
```

to see how readily a solid area can be colored with the PAINT statement.

The general form of the PAINT statement is **PAINT (x,y),paint,boundary** where **x** and **y** are the absolute or relative coordinates at which the painting is to start, **paint** is the color with which the area is to be filled, and **boundary** is the color of the border of the shape that is to be colored. Paint and boundary are optional, with paint default-

ing to the foreground color (color 3) and boundary defaulting to the paint color.

In plain English, what PAINT does is best described as: "Starting at x,y, fill in with the paint color everything that can be reached without touching or passing anything that is the boundary color."

If the start point is the boundary color, PAINT has no effect. The last point referenced by PAINT is the start point.

Note: an "Illegal function call" error results if you attempt to PAINT starting at a point off the screen, whether that point was specified by absolute or relative addressing. However, that off-screen point is the last point referenced despite the error. For instance, **PAINT (160,200)** produces an error, but if the command is immediately followed by **PSET STEP(0,-1)** a dot is drawn at (160,199). This can be important if you are "trapping" errors with the ON ERROR statement.

Let's PAINT a little. Clear the screen in medium-resolution graphics mode and type:

```
LINE (20,100)-(50,130),,B: PAINT (30,110) and  
press Enter
```

to produce a solid square. Now try:

```
CIRCLE (110,100),50,2: PAINT STEP(0,0),2 and  
press Enter
```

to produce a solidly colored ball, and:

```
CIRCLE (220,100),50,3: PAINT STEP(0,0),1,3 and  
press Enter
```

to create a colored ball with a differently colored edge. (The difference between the ball and its edge may not be clear on a television display.)

Notice that while the BF option to the LINE statement could have produced the solid square, it would have been

extremely difficult to produce the solid circles without the PAINT statement.

The object to be filled in need not be regular. Clear the screen and type:

```
CIRCLE (160,100),50: LINE (40,40)-(220,150):  
CIRCLE (180,90),40 and press Enter
```

Then type:

```
PAINT (170,80),2,3 and press Enter
```

Clearly, even the most jagged form can be filled in easily with the PAINT statement.

Clear the screen and type:

```
CIRCLE (160,100),50,3: PAINT STEP(0,0),2 and press  
Enter
```

The whole screen fills in, because the boundary color defaults to the paint color, which is 2 in this case. Because the circle is drawn in color 3, the boundary color is never found, and the painting continues to the edge of the screen. The statement **PAINT STEP(0,0),2,3** would have worked properly.

Incidentally, the above example demonstrates that PAINT is a useful, albeit somewhat slow, way to set the entire screen to a color other than color 0, the background color.

PAINTING TIPS

The use of PAINT is straightforward: if there is a figure you want filled in, start PAINT somewhere inside that figure, and if there is an area you want filled in, outline and PAINT it.

You may have noticed that PAINT goes off in each of several directions in turn when it starts in the middle of a figure or when it comes to an irregular surface. Thus, the PAINT command fills the entire area within the boundary. Starting the command in a corner (preferably at the upper or lower edge of the figure, such as the twelve o'clock point on a circle) enables PAINT to operate more smoothly.

PAINT uses a good deal of memory to keep track of all the turns it must make. With a complex figure, there can be many pending turns, and the PAINT command may run out of memory. If this occurs, an "Out of memory" error message is displayed. More memory can be allocated for the use of the PAINT command (at the expense of program memory) by entering **CLEAR,,3000**. This should take care of any potential memory problems related to PAINT while leaving plenty of memory available for your programs. Only in the case of a very large program and/or a very complex PAINT command should the amount of memory reserved in the CLEAR command be larger. Remember that the CLEAR command sets the value of all variables to zero, so it should be used only at the beginning of a program. Check the *BASIC* manual for more information on the CLEAR command.

A condition to watch for on television sets and composite monitors is the apparent presence of gaps in the boundary of a colored figure, especially when the figure is a circle. The edges are there whether the screen displays them or not, and PAINT recognizes these edges and stops at them. The gaps appear because, often, only every other colored pixel appears on composite displays. If you become confused in such a situation, use white which always shows every pixel to draw the boundary of the figure.

You may have noticed that PAINT can be slow. This is especially apparent when filling in large, irregular figures. When using PAINTed forms in animation, the GET and PUT graphics commands will increase greatly the speed of the animation. In general, though, PAINT is a remarkably useful, easily applied command that puts the splashy color graphics of the PC at your fingertips.

An Example of Paint

The program shown in Listing 8-1 demonstrates the use of the PAINT statement. Three circles are drawn and each is PAINTed a different color (see Fig. 7 in the color photograph section). The specification of STEP(0,0) as the starting point for each PAINT statement means that each

PAINT statement starts at the center of the circle just drawn. This is an easy way to make sure the PAINTing will begin within the bounds of the circle.

Listing 8-1. PAINT Statement.

```

100 REM Program to demonstrate the PAINT statement.
110 SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
120 REM Draw three circles and PAINT starting at
130 REM the centers with three different colors
140 CIRCLE (120,80),40,3      'Draw first circle
150 PAINT STEP(0,0),3,3      'Fill in with white
160 CIRCLE (150,120),40,3    'Draw second circle
170 PAINT STEP(0,0),1,3      'Fill in with blue
180 CIRCLE (180,80),40,3     'Draw third circle
190 PAINT STEP(0,0),2,3      'Fill in with purple
200 LOCATE 24,8:PRINT "PRESS ANY KEY TO CONTINUE";
210 A$=INKEY$:IF A$="" THEN 210 ELSE CLS
220 END

```

CHAPTER 9

A PIE CHART PROGRAM

We are now ready to put our knowledge to work in the design of a program. Our program will prompt the user for a number of data points, and then draw a circle sliced into wedges which are sized according to the data entered by the user. This type of display is known as a *pie chart*, and is useful for quickly communicating relative magnitudes. Pie charts are standard tools for business and statistical presentations. The complete program is shown in Listing 9-1, and a sample pie chart is shown in Fig. 8 in the color photograph section.

The overall logic flow of the program follows. First, the screen is set for graphics, then the data is input, and, finally, each wedge is drawn in turn, sized in proportion to the data it represents, and labeled appropriately. We can now turn this general outline into actual BASIC program code.

First, line 120 clears the screen and sets the screen to medium-resolution graphics mode with palette 1 selected against a black background. Next, on line 140, the program prompts the user for the number of items to be graphed. (The INPUT statement shown first prompts the user with the string "Number of items:" and then stores the value typed at the keyboard in the variable N.) The FOR . . . NEXT loop on lines 170 through 210 prompts for the label and value of each of the N data points specified previously. Also maintained in this loop is a sum of the total value of all the data points typed, so that the proportion represented by each point can later be cal-

Listing 9-1. Pie Chart.

```

100 REM Piechart program.
110 REM Initialize the screen
120 SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
130 REM Get all the values needed
140 INPUT "Number of items (1-9)";N
150 IF N<0 OR N>9 THEN 140
160 TOTAL=0
170 FOR I=1 TO N
180     INPUT "Number, Label";ITEM(I),L$
190     LABEL$(I)=LEFT$(L$,7)      'Labels 7 long maximum
200     TOTAL=TOTAL+ITEM(I)        'Sum of total of inputs
210 NEXT I
220 CLS                          'Clear the screen
230 OLDANG=0                     'Init angle at which last wedge was drawn
240 TWOPI=6.28                   'The radian equivalent of 360 degrees
250 FOR I=1 TO N                 'There will be N wedges drawn
260     REM Calculate fraction of circle taken by this wedge
270     ADDANG=TWOPI*(ITEM(I)/TOTAL)
280     REM Calculate the ending angle for this wedge
290     NEWANG=OLDANG+ADDANG
300     REM Draw the wedge with a radius of 60 dots in white
310     CIRCLE(160,100),60,3,-OLDANG,-NEWANG
320     REM Calculate the x and y components of the line which
330     REM     would go through the center of the wedge
340     XCOMP=COS(OLDANG+(ADDANG/2))
350     YCOMP=-SIN(OLDANG+(ADDANG/2))*(5/6)
360     REM Fill in the wedge by painting starting at a point
370     REM     in the center of the wedge
380     PAINT(160+XCOMP*30,100+YCOMP*30),I MOD 4,3
390     REM Find a place in text character location terms to put
400     REM     the label for this wedge
410     LOCATE (100+YCOMP*95)/8,(140+XCOMP*95)/8
420     PRINT LABEL$(I)           'Label the wedge
430     REM The new angle for this wedge is the old angle for
440     REM     the next wedge
450     OLDANG=NEWANG
460 NEXT I
470 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
480 IF INKEY$="" GOTO 480 ELSE CLS
490 END

```

culated. Note the liberal use of comments to aid comprehension of the program; this practice makes it easier to modify a program later, as well as assisting in debugging.

Line 220 clears the data input dialogue from the screen.

Because each wedge is drawn in turn, it is simplest to start drawing each new wedge at the end of the previous wedge. The variable OLDANG is the angle (measured, as

are all angles in this program, in radians) of the counter-clockwise side of the previous wedge. OLDANG is initially set to zero on line 230 so that the first wedge will begin at the three o'clock position.

The variable TWOPI equals the constant value of the number of radians in a complete circle, which is 6.28, as set on line 240.

We are now ready to begin drawing each wedge in turn. The FOR . . . NEXT loop extending across lines 250-460 does this while also plotting each wedge and coloring and labeling it.

Line 270 calculates the fraction of a full circle that the current data point represents. This fraction, stored in the variable ADDANG, is equal to the distance around the circle from the start of the wedge being plotted. On line 290, this wedge width is added to the wedge starting point to produce the wedge end point, which is stored in the variable NEWANG.

Line 310 does all the actual drawing of the wedge. The circle that the wedge is cut from is centered on the screen, has a radius of 60, and is drawn in white (color 3). Also, note that the use of minus signs ("−") in front of OLDANG and NEWANG causes a line to be drawn from each end of the arc to the center of the circle, forming the wedge.

Now we can color the wedge with PAINT. First we must figure out where to start the PAINT. The simplest approach is to start halfway between the edges of the wedge, and halfway between the center and the edge of the circle. Lines 340 and 350 calculate XCOMP and YCOMP, the x and y components of the line that extends from the center of the circle midway between the sides of (that is, bisecting) the wedge. Note that SIN is made negative; this is because mathematical convention has y increasing from the bottom upward, but, as we have already mentioned, on the PC screen y increases from the top downward. Also, the y component is multiplied by 5/6 to compensate for the aspect ratio of the screen. (Recall the discussion of aspect ratio in Chapter 7.)

Line 380 colors the wedge. White is the boundary color of the PAINT, because line 300 outlined the wedge in

white. The section reading **I MOD 4** generates the values 0,1,2,3,0,1,2,3,0,1, . . . for the color parameter over and over as the value of I increases, thus cycling through each of the four available colors in turn. Each wedge is a different color than the wedge before.

The label can now be printed by lines 410-420. The label is centered on the same bisecting line used to select the point on which PAINT began, but it is located outside the circle. The value 95 is used in line 410 so that the label is printed well outside the circle which has a radius of 60. The values are divided by 8 because the LOCATE statement, which sets the screen position at which the next PRINT occurs, works in terms of characters, not pixels, and each character is 8 pixels wide and 8 pixels high.

The wedge is now complete. Because the next wedge begins where this wedge ended, the next wedge's trailing edge (OLDANG) is set equal to the leading edge of the current wedge (NEWANG). The FOR . . . NEXT loop then proceeds to the next data point.

After all the data points have been plotted, it remains only to end the program. As discussed earlier, the INKEY\$ function returns a null string (" ") when no key has been pressed. Line 480 monitors the keyboard continually and ends the program only after a key press is detected. At this point the program is finished.

Type and run the *Pie Chart* program; for example, use 4 as the number of data points, and type the following data and labels: 5, Alpha; 10, Beta; 12, Epsilon; and 6, Gamma. Experiment with several data sets of your own, examining how the program centers labels and how it sizes and colors the wedges.

IMPROVING THE PIE CHART PROGRAM

The *Pie Chart* program we have discussed is functional as is. However, there is room for improvement. The program is not as convenient to use as it might be, and the output could be labeled more informatively.

It is inconvenient to have to type the data each time the program is run. When making a presentation, particularly

if several charts are to be drawn, it would be preferable to run the program from previously typed information. An alternative is to place the data in a disk file first, then read the data from the file and into the program. Then, all that needs to be typed when the program is run, is the name of the disk file. The data can be put into the disk file either with an editor (such as IBM's *Edlin*[™]) or a word processor (such as *WordStar*[®]), or with a separate BASIC program that prompts for the data in much the way that our program does and then writes the data to the disk file. The information necessary to work with disk files in BASIC is contained in the *BASIC* manual explanation of the OPEN, CLOSE, READ #, and WRITE # statements.

It would be desirable to have an identifying title at the top of the chart. The pie chart might be more effective if there were pointers from the labels to their corresponding wedges. The percentages or absolute values associated with each of the wedges could also be placed next to the labels. The object is to convey the maximum amount of information to the viewer as easily as possible. The pie chart visually indicates relative magnitudes; additional information can be available on the screen should the viewer desire to know more.

Consideration of the program user is the paramount concern in designing such a program. The colors and graphics serve only to convey information, and have no function in and of themselves. The data input should be as easy as possible, both for the novice and for the frequent user. Never forget that while it is satisfying to design dazzling graphics, software that is engineered for the user is more likely to be used than software which does not consider the user. If you develop your skills in this area, even simple programs can be appealing and practical to use.

Apple and Applesoft are registered trademarks of Apple Computer Inc. Edlin is a trademark of International Business Machines Corporation. WordStar is a registered trademark of Micropro International Corporation.

CHAPTER 10

ANIMATION FROM BASIC—THE GET AND PUT STATEMENTS

Animation is the process of rapidly drawing and redrawing a figure to create the impression of motion. Of course, there really is no motion, but, like movies, the rapid succession of images fools the eye. All arcade games involve animation, and animation is useful in other applications for attracting and guiding viewers' attention.

The graphics statements discussed thus far give us the ability to draw figures of virtually any kind; but when it comes to animation, statements such as LINE and CIRCLE are nothing more than tools because of their lack of speed. These statements can construct an image but cannot manipulate that image rapidly enough. What we need is a way to save a previously drawn image, then restore it rapidly to the screen. Happily for us, the GET and PUT graphics statements do precisely that.

For example, clear the screen in medium-resolution graphics mode and type:

```
DIM B(100): CIRCLE (160,100),10: GET  
(145,85)-(175,125),B and press Enter
```

then clear the screen and type:

```
FOR I=10 TO 250 STEP 10: PUT (I,100),B: PUT  
(I+10,100),B: NEXT and press Enter
```

and you've produced your first animation! Such a program would be impossible with the slow CIRCLE state-

ment—but it is a “snap” with the GET and PUT graphics statements.

Note that there are also GET and PUT statements related to random-access disk-file operations. These statements are unrelated to the GET and PUT graphics statements. When we refer to GET and PUT in the remainder of this book, we mean the GET and PUT graphics statements.

The following background information may help your understanding of GET and PUT. An *array* is a series of variables strung together, all referred to by the same variable name but with different index numbers known as *subscripts*. For example, **DIM TEMP(100)** sets aside 101 storage locations to be referred to as TEMP. The first location is referred to as TEMP(0), the next as TEMP(1), and so on up to TEMP(100). All the storage locations referred to by TEMP constitute the array TEMP.

The GET graphics statement transfers the contents of a rectangular area of the screen into an array. The general form of the statement is **GET (x1,y1)–(x2,y2),arrayname** where **x1**, **y1**, **x2**, and **y2** are the coordinates of opposite corners of the rectangular area and **arrayname** is the array into which the contents of that area of the screen are to be stored. The coordinates are specified precisely, as are the coordinates for the LINE statement, with either coordinate in absolute or relative form. However, the first coordinate is not optional. In fact, all of the parameters are required. The last point referenced by GET is the point (x2,y2).

The array may be of any numeric data type, but it is easiest to use arrays of type integer consistently. The array must be dimensioned by the DIM statement large enough (have enough memory locations set aside) to contain the specified screen area. For integer arrays the form for array size is $\text{INT}((x/8)+1)*y+2$ where $x=\text{ABS}(x2-x1)+1$ and $y=\text{ABS}(y2-y1)+1$. This will calculate roughly the required size of the array in medium-resolution graphics mode, leaving a little extra space just to make sure. (The $z=\text{ABS}(x)$ function returns the absolute value of

x. If x is negative, then $z=x*(-1)$; if x is positive, then $z=x$.)

For example, if you were to enter **GET (10,10) - (19,29), PIC1** in medium-resolution mode, the integer array PIC1 would have to be dimensioned to **INT ((10/8)+1)* 20+2 = 42** storage locations using the statement **DIM PIC(42)**. The GET statement section of the *BASIC* manual describes the method for calculating the exact minimum allowable size of the array, but, unless you're running out of memory, it's easiest to make a rough estimate on the safe side.

If the array is, in fact, too small to hold the image, or if the GET statement references a point off the screen, an "Illegal function call" error results when the GET is performed. If this happens, the last point referenced is still set. Precisely what point that will be depends on what error occurs, so be aware that the last point referenced may behave unexpectedly if the GET statement generates an error.

The PUT graphics statement is the companion statement to GET but is not strictly the reverse of GET. In fact, there are five ways in which the image may be PUT into the screen, and each has a different application.

The PUT statement is **PUT (x1,y1),arrayname,action**. Coordinates **x1** and **y1** define where the upper-left corner of the image is to be placed; **arrayname** is an array in which an image has been stored; and **action**, the only optional parameter, selects one of the five ways in which the image may be placed onto the screen. Valid values for action are **PSET**, **PRESET**, **XOR**, **OR**, and **AND**. If no value for action is selected, **XOR** is the default. The coordinates may be specified in absolute or relative form. The last point referenced is (x1,y1). As with GET, an "Illegal function call" error results if PUT references a point off the screen. The last point referenced is (x1,y1) even if an error occurs.

THE FIVE PUT OPTIONS

The most useful of the five options with which an image can be PUT onto the screen are **PSET** and **XOR**. The **PRE-**

SET, OR, and AND options have few practical applications and often result in undesirable color effects. Fig. 9 in the color photograph section, which is produced by the program in Listing 10-1, shows the operation of the five options against various backgrounds.

PUT with the PSET option is nothing more than the reverse of GET. The image previously stored in an array with the GET statement is then displayed on the screen with **PUT (x,y),arrayname,PSET**, erasing what was already there. The PSET option is useful for copying an image drawn with the PSET, LINE, CIRCLE, and PAINT statements to a new screen location without laboriously redrawing the image. The PSET option is also useful for certain types of animation.

The PRESET option is similar to the PSET option but puts the negative of the image onto the screen. In medium-resolution mode colors 0 and 3 are reversed, and so are colors 1 and 2. That's how the PRESET option works; in all honesty, it's easier to describe the PRESET option than it is to imagine a use for it.

The XOR option is the default for the PUT statement. The operation of the XOR option causes the pixels in the image and the corresponding pixels in the screen to be logically eXclusive ORed (XORed) together. The operation of the XOR option, along with that of the other four options, is explained in more detail in the section of the BASIC manual that covers the PUT statement.

The XOR option is often used to PUT images against the background color; in this case, the effect of one PUT with the XOR option is to draw the image on the screen, and the effect of another PUT with the XOR option with the same image in the same location is to erase the image entirely. Therefore, the XOR option can be used to erase an image, as well as to draw it. For this reason and others to be discussed subsequently, the XOR option is useful for animation.

The OR option to the PUT statement superimposes an image over whatever image is already on the screen. Unlike the PSET and PRESET options, the portions of the image to be PUT that are the background color (color 0) do

not erase the previous image. Basically, the second image is placed in front of the original.

The AND option puts an image onto the screen only where there's already an image. While we undoubtedly could come up with some use for the AND option, the truth is that the AND option is not especially useful.

For a demonstration of the five PUT options, type and run the program in Listing 10-1. This program puts an image onto the screen against each of the four colors. Note the various color and background effects produced. For more details on the various options, refer to the tables under the PUT statement in the *BASIC* manual.

Listing 10-1. PUT Statement Options.

```

100 REM Demonstration of the five options to the
110 REM PUT statement.
120 REM Initialize screen
130 SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
140 REM Draw concentric circles of different colors
150 CIRCLE(25,25),25,3:PAINT STEP(0,0),3
160 CIRCLE(25,25),20,2:PAINT STEP(0,0),2
170 CIRCLE(25,25),15,1:PAINT STEP(0,0),1
180 CIRCLE(25,25),10,0:PAINT STEP(0,0),0
190 REM Save the image of the circles just drawn
200 DIM IMAGE(364):GET(0,0)-(50,50),IMAGE
210 REM Clear the screen
220 CLS
230 REM Put in labels for the five options
240 LOCATE 14,1
250 PRINT "****PSET***PRESET**XOR*****AND*****OR****"
260 REM Set up a background of different colors so that
270 REM the various color effects can be observed
280 FOR I=0 TO 62
290 LINE(I*5,120)-(I*5+5,180),I MOD 4,BF
300 NEXT I
310 REM Draw the saved image on a black background and
320 REM on the colored background using each of the
330 REM options to the PUT statement
340 PUT(20,125),IMAGE,PSET:PUT(20,50),IMAGE,PSET
350 PUT(80,125),IMAGE,PRESET:PUT(80,50),IMAGE,PRESET
360 PUT(140,125),IMAGE,XOR:PUT(140,50),IMAGE,XOR
370 PUT(200,125),IMAGE,AND:PUT(200,50),IMAGE,AND
380 PUT(260,125),IMAGE,OR:PUT(260,50),IMAGE,OR
390 LOCATE 25,9:PRINT "PRESS ANY KEY TO CONTINUE";
400 IF INKEY$="" GOTO 400 ELSE CLS
410 END

```

ANIMATION

As described in a preceding section, PUT and GET provide us with the capability to produce animation. There are rules that must be followed in order to produce convincing animation. First, the image on the screen must be moved only a short distance each time so the motion seems continuous. Also, the previous image must be removed from the screen; if it is not, the moving image will appear to leave a trail in its wake.

The animation program in Listing 10-2 repeatedly PUTs an image on the screen with the PSET option and then PUTs a blank area to erase the image before redrawing it in a new location. Lines 140 through 160 draw a colored ball, using the CIRCLE statement, and line 180 saves the image in the array BALL. Line 200 GETs a blank area of the same size, while lines 250 through 280 create the impression of movement by drawing, erasing, moving, and redrawing the ball across the screen.

Listing 10-2. PUT Statement with PSET Option.

```

100 REM Demonstrate animation using the PUT statement with
110 REM the PSET option to repeatedly draw and erase.
120 SCREEN 1,0:COLOR 0,1:KEY OFF:CLS 'Set med-res screen
130 REM Draw a concentric colored ball
140 CIRCLE(12,12),8,3:PAINT STEP(0,0),1,3
150 CIRCLE(12,12),6,3:PAINT STEP(0,0),2,3
160 CIRCLE(12,12),2,3:PAINT STEP(0,0),3,3
170 REM Save the image of the ball
180 DIM BALL(144):GET (0,0)-(23,23),BALL
190 REM Save a blank area the size of the saved ball image
200 DIM BLANK(144):GET (50,50)-(73,73),BLANK
210 REM Clear the ball off the screen
220 CLS
230 REM Repeatedly erase and draw ball at slightly changed
240 REM positions to create the illusion of motion
250 FOR X=8 TO 288 STEP 2
260 PUT (X-2,60),BLANK,PSET:PUT (X,60),BALL,PSET
270 FOR I=1 TO 30:NEXT I 'Wait to minimize flicker
280 NEXT X
290 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
300 A$=INKEY$:IF A$="" THEN 300 ELSE CLS
310 END

```

Animation is the basis for all video games. Arcade games and some computers use specialized hardware to

produce animation, but because the PC is a general-purpose computer, it has none of this equipment, so animation must be produced by the software. In BASIC, we use the PUT statement to animate.

Animating with XOR

The XOR option is the simplest way to produce animation on a complex screen, and, indeed, eXclusive ORing is used in most games found on home computers. The great virtue of XOR is that it can be used to draw and then erase an image without changing the background, even if several images are moving past each other at the same time. By way of contrast, the PSET option always erases the background of the PUT.

When erasing and redrawing the image being animated, a distracting flicker can occur. One PUT erases the old image, and the next PUT draws the new image; during the interval between the two PUTs, there is no image on the screen, thus causing a flicker. The longer this interval, the worse the flicker, so the two PUT statements should, if possible, occur without intervening operations.

When a figure created with the XOR option is drawn over another figure, some odd color effects may occur where the two images overlap. This effect is normal and disappears when the overlap ceases. As mentioned previously, odd color effects can occur with all PUT options except PSET.

The effects of the XOR option to the PUT statement in medium-resolution mode are shown in Table 10-1.

Table 10-1. GET and PUT with XOR

PUT Color	Background Color			
	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

The program shown in Listing 10-3 demonstrates animation using PUT with the XOR option. Notice the color

interaction, the flicker of the moving ball, and the preservation of the background. The program pauses briefly after drawing each image so the image is on the screen longer than it is off. In this way, the flickering of the image is minimized.

Listing 10-3. PUT Statement with XOR Option.

```

100 REM Program to demonstrate animation with the
110 REM   XOR option to the PUT statement.
120 REM Initialize screen
130 SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
140 REM Draw a colored ball
150 CIRCLE(12,12),8,3:PAINT STEP(0,0),1,3
160 CIRCLE(12,12),6,3:PAINT STEP(0,0),2,3
170 CIRCLE(12,12),2,3:PAINT STEP(0,0),3,3
180 REM Save the image of the ball
190 DIM BALL(144):GET(0,0)-(23,23),BALL
200 REM Clear the ball from the screen
210 CLS
220 REM Draw a series of colored stripes for ball
230 REM   to move across
240 FOR I=4 TO 12
250   LINE(I*20,50)-(I*20+19,100),I MOD 4,BF
260 NEXT I
270 REM Put the image on the screen in its first
280 REM   location so that the next PUT will
290 REM   erase it
300 PUT(6,60),BALL,PSET
310 REM Repeatedly erase and then draw the ball
320 REM   moving it 2 pixels right each time
330 FOR X=8 TO 288 STEP 2
340   PUT(X-2,60),BALL,XOR:PUT(X,60),BALL,XOR
350   REM Wait to minimize flicker
360   FOR I=1 TO 30:NEXT I
370 NEXT X
380 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
390 A$=INKEY$:IF A$="" THEN 390 ELSE CLS
400 END

```

Animating with PSET

When we produced animation earlier with the PSET option to the PUT statement, we PSET the image initially, PSET a blank area of equal size over the image to erase it, then redrew the image in a new location. This particular method is, in general, inferior to using the XOR option, but there is another, better way to use PSET for animation.

To use PSET for animation, we must first GET an image with a border that is wider than the distance the image appears to jump each time it is moved, then PUT the image onto the screen with the PSET option. Each time the image is redrawn with the PSET option, the blank border will erase the previous image automatically. (Remember that the image cannot move any farther than the width of the border in any single move.) There is never any separate operation to erase the image. Because an image is always on the screen, the animation produced is smooth and relatively flicker-free. Also, because there is only one PUT performed each time the image is moved, less time is used in drawing, and so the action can move faster.

The program shown in Listing 10-4 demonstrates animation with the PSET option. Type and run it, noting that the moving ball now flickers less than when it was animated with the XOR option.

You will also see that the PSET option does not preserve the images on the screen as it passes over them. The background can be restored with appropriate redrawing in the wake of the moving object, but this is difficult to do. The XOR option, on the other hand, does not require redrawing the background. If there is no background landscape, the PSET option works well; if there is a landscape, the PSET option can cause more problems than it is worth. Care must also be taken that the blank border of the image does not go off the screen, as this will cause an error even though the figure appears to be fully on the screen.

Selecting between the XOR and PSET options to the PUT statement depends on the complexity of the animation task. In general, the XOR option is simpler and more widely used.

FUN AND GAMES

If you plan to write your own video games, study this chapter carefully, not only for the BASIC statements, PUT and GET, but for the basic principles of animation. Understanding the concept of the eXclusive OR (XOR) method of

Listing 10-4. PUT Statement with PSET Option and Border.

```

100 REM Program to demonstrate animation using the PUT
110 REM statement with the PSET option and a border
120 REM around the image to erase the old image.
130 REM Initialize the screen
140 SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
150 REM Draw a ball
160 CIRCLE(12,12),8,3:PAINT STEP(0,0),1,3
170 CIRCLE(12,12),6,3:PAINT STEP(0,0),2,3
180 CIRCLE(12,12),2,3:PAINT STEP(0,0),3,3
190 REM Save the image of the ball and some of
200 REM the black border
210 DIM BALL(144):GET(0,0)-(23,23),BALL
220 CLS 'Remove ball from screen
230 REM Draw a series of colored bars for ball to
240 REM move across
250 FOR I=4 TO 12
260 LINE(I*20,50)-(I*20+19,100),I MOD 4,BF
270 NEXT I
280 REM Repeatedly move the ball right by 2 pixel
290 REM positions allowing the black border to
300 REM wipe out the part of the old ball image
310 REM not covered by the new image
320 FOR X=8 TO 288 STEP 2
330 PUT(X,60),BALL,PSET
340 FOR I=1 TO 30:NEXT I 'Delay a bit
350 NEXT X
360 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
370 A$=INKEY$:IF A$="" THEN 370 ELSE CLS
380 END

```

animation is particularly important. Take time now to review the material on animation before proceeding to the next chapter, in which we design a complete video game.

BLOCKBUSTER—AN ARCADE- STYLE GAME

You have now learned enough of the graphics elements of the BASIC language to draw forms of various shapes, sizes and colors, and to do so at high speed. In short, you are ready to design and program an arcade-style game. Thanks to the graphics capabilities of IBM BASIC, you'll find our *Blockbuster* game surprisingly compact and easy to implement.

Up to now, we have presented sample programs as finished products, because we have been concentrating on individual commands. In this chapter, you will begin to learn how to assemble the commands into a program designed to perform a specific task. Therefore, we will take you through the complete design process of *Blockbuster*, from conception to completion. Fig. 10 in the color photograph section shows the finished product that is our goal.

GAME DESIGN

The most important part of producing a game is creating a good design for it. The key to game design is to realize that while there are general guidelines to good design, there are no “cookbook” rules. In other words, no one can put a series of steps down on paper that, if followed correctly, will yield a functioning game, let alone an enjoyable one. Each game is an individual work, not unlike a novel or movie. The author's imagination is the critical element.

What we will do here is take you through the step-by-step construction of a game, outlining a general approach to help you to avoid frustrating dead ends, bugs, and non-functional programs.

As programs grow more complex, it becomes impossible to program "on the fly," simply writing code until the program is done. The "top-down" design we use ensures that each section of the program works well with the rest. A good design creates a logically composed program which, in turn, makes programming, debugging, and later additions and changes much easier.

The first step in the game design process is to describe the game in English. It can be useful to sit down with a pencil and paper and write a synopsis of the game. Then make an outline detailing how the program will accomplish each element of the game. Before writing a single line in BASIC, you should determine specifications for the playing field and moving objects, and decide how to handle the interactions among all components of the game. Once these decisions are made, the program can be written easily from the specifications, without any need to worry about overall program logic.

BLOCKBUSTER

The game we will design is called *Blockbuster*. It is a "pong"-type game in which a "ball" directed by a player-controlled paddle bounces around a playing field and into a wall of "bricks." Each time the ball hits a brick, that brick vanishes. The game ends when the player clears the field of bricks or runs out of balls.

We have just completed our synopsis of *Blockbuster*. There are five sections of the program to be designed before the game can be programmed. These are the playing field, the bricks, the paddle, the ball, and housekeeping (referring to the section that begins, runs and ends a program "neatly.")

The Playing Field

The playing field is closed on all four sides, but the side located at the bottom of the screen functions as an essen-

tially open side in that, when the ball reaches this area, it is deemed to be out of play and thus lost. The paddle is on this open side. Putting the paddle at the bottom creates the illusion of gravity, because the ball will appear to be falling toward earth as the player hurries to intercept its path. Certainly the game would not seem as real if the paddle were at the top and the ball appeared to fall upward!

The playing field is nearly square at 166 rows by 162 columns (all coordinates are measured in pixels). Here we have our first encounter with the inexact nature of game design. There is nothing sacred about the dimensions we've selected, and any number of other dimensions would serve as well. A number of factors go into such a decision, including screen size, brick size, number of bricks, ball speed, and general playability of the game. We have selected our size as reasonable and workable, but, often in the game design process, there are several iterations of the design process as the design closes in on one that the designer "feels" is right (there being no one "right" design). If you wish to experiment with the playing field size (or, indeed, any aspect of the game) please do, but remember that you must consider the effects on other interacting parts of the game. This is why design is best done before the programming phase; imagine programming the playing field size, only to arrive at the program section that draws the bricks and discover the bricks will not work with the playing field you've created!

The playing field is black, as that is the easiest background color on which to follow the action. The area surrounding the playing field is red.

Color selection for graphics is always a problem on the PC, because in medium-resolution graphics mode, we only have four of the sixteen colors available, and then only the colors in one of the two palettes. We have already decided that we will use black, and because black is not in either of the palettes, it must be the background color. Now we must select our palette. As discussed in Chapter 3, palette 0, with red, green, and brown, is vivid on a high-quality monitor, but all three colors look brownish on a tv screen.

Palette 1 has magenta, white, and cyan, which are less striking but look acceptable on all screens.

The nature of *Blockbuster* is such that either palette is fine; we will use palette 1 because it produces a good display on any screen. In this case, the game did not dictate the colors, but this is not so in many cases. For example, a game program which used trees and other plants in the background would certainly require green to represent foliage, so palette 0 would be preferable.

We will use color 2, magenta, for the area surrounding the playing field, because this is an attractive but relatively subdued color. By contrast, white would not be a good choice for the border, as it would distract from the game and tire players' eyes. Note that magenta (which is a light red) may appear as any color from pink to deep red on various displays.

The Bricks

Once the playing field is set, the bricks can be drawn on it. There are four rows of eight bricks each, arrayed about two-thirds of the way to the top of the playing field. Each brick is 9 rows by 19 columns in dimension.

Again, there are no rules regarding placement and sizing. The usual approach is to use graph paper, a pencil, and a large eraser to experiment, considering the size of the playing field and the nature of the game as well as the bricks themselves. (All the dimensions of the game are usually determined during one trial-and-error process. When various objects interact, none of them can be designed in isolation, so one may go through several tentative sizings before arriving at the final dimensions.)

The upper edge of the top row of bricks begins at row 48, as counted from the top of the screen. Each brick is 3 rows below the brick above and 1 column from each brick to the side. The bricks are alternating cyan and magenta.

The Paddle

The paddle is a horizontal white line 21 columns wide, set on row 181 of the screen. The paddle's range of movement

is the full width of the playing field, from columns 80 to 241.

Though a joystick is an ideal paddle controller, not all PC owners have joysticks. Therefore, *Blockbuster* uses the keyboard to control the paddle, even though joystick control would be easier.

One possible method of keyboard control involves moving the paddle one unit each time a certain key is pressed. With this method approximately 25 keystrokes are required to move the paddle across the screen. Because the PC keyboard doesn't repeat a given key immediately (i.e., continually send the same key value when that key is held down), this method would force the player to "whack" the keyboard with alarming frequency.

A better approach to keyboard-controlled movement is to use a specific key press to start the paddle moving and a second key to stop the paddle movement. (The paddle would stop moving when it appears to touch the edge of the playing field as well.) In *Blockbuster*, the z key starts the paddle moving left, the x key stops movement, and the c key starts the paddle moving right. (All these keys are lowercase.) Notice that these keys are grouped together so the left hand can conveniently control the paddle. Of course, you may select different keys.

It would be a nice touch if we could handle both uppercase and lowercase characters, so the shift state would be irrelevant. What would be even better would be a message that briefly halted play and informed the player if the key struck was not a valid key. This would, however, require that the IF statement that interprets keystrokes be longer and slower, so we elected to implement neither of these features.

The Ball

Handling the ball is the most complex portion of the program, because the ball moves in two dimensions and interacts with each of the elements previously discussed. The ball is a solid circle of radius 2, starting in the middle left of the screen below the bricks and aimed directly at the paddle. This makes the game easier to play—if a

player is confused or unprepared, the ball won't be lost immediately. Also, when the ball bounces off the paddle at the start of the game, the basic idea of the game becomes readily apparent to the novice.

To create the impression that the ball is moving, the program erases the ball at its current location, adds to the ball's coordinates an increment in each of the x and y directions (x being horizontal and y, vertical), and then redraws the ball in the new location. (At this point, you may wish to review the discussion of animation in Chapter 10.) The x and y increments fully describe the motion of the ball, and each can be either positive or negative to describe movement in all directions. A positive x will cause the ball to move right and a positive y will cause the ball to move "down," with negative values causing the opposite motions.

To determine if the ball has struck any of the edges of the playing field, we compare coordinates. If the ball is at column 80, for example, and the computer tries to move the ball to the left (by addition of a negative x increment), it has hit the left edge. In this case, the x increment must be negated (multiplied by -1) to reverse motion away from the left wall.

We similarly compare coordinates to check whether the ball has hit the paddle or gone off the bottom of the screen. If the ball reaches the paddle row, it either hits the paddle or misses the paddle and is "lost."

If the ball "hits" the paddle, the y increment is reversed to move the ball back up, and the x increment is set to a value that varies according to how close to the center of the paddle the ball strikes.

Why not simply leave the x increment the same, so the ball "flies off" at the same angle that it approached? Varying the path of the ball as it bounces off the paddle makes the game more varied and interesting. This also creates the impression that the ball and paddle are real objects; in reality, balls bounce with spin and at many angles, so it is desirable to create the illusion that this is happening in our game. Details such as this make the game feel "right"

to players, even though they may not know why they have that feeling.

If the ball reaches the paddle row and does not strike the paddle, it is “lost”: The ball is erased, then restarted as it was at the beginning of the game. The task of counting lost balls is handled in the housekeeping section.

The program checks the screen to see if the ball has hit a brick. If the ball hasn't hit the edge of the playing field or the paddle, and if the pixel located where the center of the ball is about to move is not black (the background color), then the ball must have encountered a brick. Disposing of the brick is simple; PAINT it off the screen, starting at the point just checked.

In considering the ball's motion and the collision checks, it is important to understand the nature of program execution. While you, the game designer, are trying to create the impression of continuous action, you are dealing with discrete, sequential events, because BASIC commands can only do one thing at a time. Hence, in *Blockbuster*, you know that any point not black must be a brick only because you have already eliminated the possibility that the point is the edge of the playing field or the paddle. If the order of checks is changed, the program will not work. Similarly, you must PAINT off the brick after the ball is taken off the screen at the old location and before it is drawn at the new, or you will PAINT the ball off the screen as well.

The discrete nature of video game programming is one reason for doing extensive game design before programming; events must occur in sequence so they interact properly and without interference. It's impossible to replicate real-world action perfectly, but good design produces good illusion. Fortunately, fast action and the impressionable eye often cover for a good deal of imprecision.

Housekeeping

Most games have a beginning, an end, and a scoring method. In *Blockbuster*, the score is the number of bricks remaining, and the end comes either when three balls are

lost or when all the bricks are gone. In the former case, players lose; in the latter, they win. In both cases, the player is notified.

The beginning of *Blockbuster* consists of initializing and drawing the playing field, bricks, paddle, and ball, with the ball's motion increments set for movement toward the paddle. Note that parts of housekeeping have been discussed in other sections. This is because the other sections would not function without these elements. For example, the ball section could not be tested if the ball were not given an initial location. We need to be able to properly test each section before proceeding to the next.

Finally, housekeeping includes counting the balls lost and bricks hit, so the progress of the game can be monitored. This information is displayed at the top of the screen, away from the playing field area.

This completes the specification of *Blockbuster*; at this point we can proceed to programming the game in BASIC.

PROGRAMMING THE GAME

Now that *Blockbuster* is designed, we could sit down with the specifications and start writing, but it is more productive to program in modules (i.e., to write each section in turn, thoroughly testing and debugging it before proceeding to the next section). This is especially important with large programs. Modular programming is easier to do with compiled languages or in Assembly language, where each module can be physically separated from the rest in the program, but it can be applied to interpreted BASIC as well.

Blockbuster is programmed in five steps, with a pause after each step to test our progress to that stage. These five steps correspond to the five sections of the program: playing field, bricks, paddle, ball, and housekeeping.

Before starting, note that the program listings shown seem to have gaps in the line numbering. For example, the line numbers in Listing 11-2 run 100, 130, 140, skipping 110 and 120. The missing line numbers are reserved for lines dealing with later steps, such as housekeeping. We

reserve these lines for explanatory purposes only. You need not leave reserved lines in your own programs, for the operation of the program does not require regular numbering; besides, any irregular line numbering can be corrected easily with the RENUM statement.

Also note that there are often many statements on one line, which can make the program difficult to read. As an interpreted language, BASIC is not particularly fast, and speed is an important consideration in program design. Statements run faster if they're on the same line, so we use multistatement lines strictly due to speed considerations. Similarly, there are no comments in the listings because BASIC wastes a surprising amount of time when it encounters REM statements and single-quote comments. In general, though, inserting comments is a desirable programming practice, and we include a listing of *Blockbuster* with complete comments at the end of this section.

A final point: as described earlier, we format the program listings with wrap occurring between words on long lines. This makes the listings more legible. Do not try to wrap your lines at the same place we did in this chapter. Several lines as shown are too long for BASIC, and will not work properly. Instead, enter all lines by typing the commands without regard for where the wrap occurs.

Programming the Playing Field

The program shown in Listing 11-1 sets up the playing field. Line 100 clears the screen and selects medium-resolution color graphics mode, with palette 1 on a black background. The DEFINT statement makes all variables integers, unless otherwise specified. (We will not specify otherwise.) This is important in programming games because BASIC variables normally default to real numbers, which are processed much more slowly than integers, particularly where arithmetic operations are concerned. The first LINE statement on line 130 colors the entire screen magenta, while the second LINE statement makes the playing field area black. Type and run the program to see the constructed playing field.

Listing 1-1. Blockbuster—Playing Field.

```
10 REM Blockbuster--playfield.  
100 DEFINT A-Z:SCREEN 1,0:COLOR 0,1:KEY OFF:CLS  
130 LINE(0,0)-(319,199),2,BF:LINE(80,20)-(241,185),0,BF
```

Programming the Bricks

Listing 11-2 adds the bricks to the playing field. In line 140 the outer FOR . . . NEXT loop with the I variable runs through the eight columns of bricks, and the inner FOR . . . NEXT loop with the J variable runs through the four rows of bricks. The heart of line 140 is the LINE statement which draws a solid brick. The coordinates of the upper-left corner of each brick are calculated as an offset from the upper left brick located at coordinates (82,48). Each column is 20 dots wide, so the x coordinate of any brick is $82 + I * 20$, where I is the column number of the brick. Likewise, each row is 12 dots high, so the y coordinate of any brick is $48 + J * 12$, where J is the row number of the brick.

For each brick, the rectangle drawn with the LINE statement is 19 columns by 9 rows and is created using STEP relative to the upper-left corner of the brick. (Note that this is an excellent example of the usefulness of relative addressing.)

The bricks are filled in with either cyan or magenta in alternating columns and rows to produce a checkerboard pattern. The MOD function is used to generate the values of the color parameter that color the bricks appropriately. The MOD function returns the remainder of the division of one number by another, so that the formula $(I + J) \text{ MOD } 2$ returns a value of 0 if (I+J) is even, and a value of 1 if (I+J) is odd. Because cyan and magenta are colors 1 and 2, we add 1 to the value of $(I + J) \text{ MOD } 2$. Therefore a MOD 2 value of 0 produces cyan and a MOD 2 value of 1 produces magenta. This formula creates the desired checkerboard pattern of brick coloring.

Run the program in Listing 11-2. Notice that it simply adds line 140 to the program in Listing 11-1.

Listing 11-2. Blockbuster—Bricks.

```

10 REM Blockbuster--playfield and bricks.
100 DEFINT A-Z:SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
130 LINE(0,0)-(319,199),2,BF:LINE(80,20)-(241,185),0,BF
140 FOR I=0 TO 7:FOR J=0 TO 3:LINE(82+20*I,48+J*12)-
    STEP(18,8),((I+J) MOD 2)+1,BF:NEXT J:NEXT I

```

Programming the Paddle

Now comes the action. The program in Listing 11-3 initializes the paddle so the player can move it back and forth. The program also sets up the playing field and the bricks, as shown previously.

Line 160 initializes the paddle's x coordinate which is stored in the variable PX. The paddle's y coordinate never changes, so the constant 181 is used throughout the program.

Lines 190, 200, and 210 control paddle movements. Line 190 uses INKEY\$ to get keyboard input, if any, without disturbing the display. (Keystrokes obtained with INKEY\$ are not shown on the screen.) If there is no input, INKEY\$ returns an empty string and nothing is done. If the input is one of the three movement keys: z, x, or c (remember, the keys must be lowercase), action is taken. The action is to set the variable PXINC (the distance by which the paddle moves each time through the loop) to -5 to move left, 0 to stop, or 5 to move right. Moving the paddle fewer than five dots at a time produces too-slow motion, while moving it more than five dots produces jerky motion that is visually distracting and is difficult for the player to control.

Line 200 guards against movement of the paddle off the edge of the playing field. Here the current paddle location is saved in OLDPX; the paddle is then tentatively moved by adding the paddle movement increment PXINC to the current paddle x coordinate, PX. If the result of the trial move is off either edge, the paddle's x coordinate is set back to the old location and no move takes place; otherwise PX is left at the new value and the new location is set.

Line 210 moves the paddle. (If PX equals OLDPX, either because PXINC is 0 or because the playing field edge has

been reached, no move occurs. Instead, the paddle is erased and drawn again in precisely the same spot. In this way, the paddle drawing always takes the same amount of time, so the game runs at the same speed whether the paddle is moving or not.) The first LINE statement in line 210 uses OLDPX to erase the paddle at its old location, and the second LINE statement uses PX to draw the paddle at its new location.

The lines we've discussed so far move the paddle once. To handle the paddle properly we need a loop that repeatedly executes the paddle movement commands. Line 270 performs this looping function. The beginning of the loop must be selected carefully. If line 160 were inside the loop, the paddle would constantly be returned to its initial position. On the other hand, lines 190, 200, and 210, which are the paddle movement commands, must be inside the loop or the paddle would never move. In fact, we loop back to line 190.

Run the program shown in Listing 11-3. Notice that the paddle moves rapidly. This is to be expected, because the lines dealing with the ball's movement are not being executed yet. These lines slow the action down considerably. There isn't any way to terminate the program yet, so use Ctrl-Break to stop when you've seen enough.

Listing 11-3. Blockbuster—Paddle.

```
10 REM Blockbuster--playfield, bricks, and paddle.
100 DEFINT A-Z:SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
130 LINE(0,0)-(319,199),2,BF:LINE(80,20)-(241,185),0,BF
140 FOR I=0 TO 7:FOR J=0 TO 3:LINE(82+20*I,48+J*12)
    -STEP(18,8),((I+J) MOD 2)+1,BF:NEXT J:NEXT I
160 PX=150:PXINC=0
190 A$=INKEY$:IF A$="c" THEN PXINC=5 ELSE IF A$="z" THEN
    PXINC=-5 ELSE IF A$="x" THEN PXINC=0
200 OLDPX=PX:PX=PX+PXINC:IF PX<80 OR PX>221 THEN PX=OLDPX
210 LINE(OLDPX,181)-(OLDPX+20,181),0:
    LINE(PX,181)-(PX+20,181),3:OLDPX=PX
270 GOTO 190
```

Programming the Ball

The program in Listing 11-4 adds the ball to what we've already created. To do this, we must draw and initialize

the ball, check for collisions with the edges, paddles, and bricks, and then move and redraw the ball.

The ball must first be drawn and saved for future use. Line 110 creates a solid white circle of radius 2, and line 120 uses the graphics GET statement to save the image of the ball into the array variable BALL.

Line 180 sets the ball's coordinates (BX and BY) and motion (BXINC and BYINC). BX and BY are set to start the ball on the left side of the playing field below the bricks. BXINC and BYINC are set initially to move the ball down and to the right to intersect the paddle's starting position. The graphics PUT statement at the end of line 180 draws the image of the ball at its starting spot. (Remember that the PUT statement is a rapid way to draw an image previously saved with a GET statement.)

Line 220 moves the ball horizontally and checks whether it has collided with either the left or right side of the playing field. The current location is stored in OLDBX for later reference. BXINC (the horizontal motion), is then added to the current x coordinate. If this new location is off either edge, the horizontal motion variable BXINC is negated (multiplied by -1), thus reversing the ball's direction, and twice the new motion is added to the x coordinate. This doubled motion both cancels the trial move made at the beginning of line 220 and moves the ball one "normal" distance in the new direction. The end result of line 220 is that the ball's x coordinate is altered by the motion stored in BXINC.

When collision checks are made, remember that BX and BY are the coordinates of the upper-left corner of the ball. Because the ball is five dots wide, its right edge is at $BX+4$. Likewise, the right edge of the paddle is at $PX+20$.

Line 230 is similar to line 220, but it handles vertical rather than horizontal motion. Only collision with the top edge of the playing field is checked, because the bottom edge and the paddle are checked for as a special case. If the ball has reached the top of the playing field, BYINC (the y motion) is reversed in the same manner as is BXINC in line 220.

Line 240 checks whether the ball has arrived at the bot-

tom edge of the playing field, which is the paddle row. If the ball hasn't reached the paddle row, no check is necessary. If the ball has reached the paddle row, it has either struck the paddle and bounced or missed the paddle and gone off the screen. If the ball hits the paddle, the ball's vertical motion (BYINC) is reversed just as when the ball strikes the top edge of the playing field. The horizontal motion is also set but is set to a greater motion the farther from the center of the paddle the ball hits, thereby created the impression of spin. The last statement on line 240 sets this variable horizontal motion.

If the ball misses the paddle, it is "erased" from the screen and restarted. This is done on line 290. Note that line 170 (where the ball is restarted after being lost) is a REM statement. This line, like line 280, is used in the housekeeping section, and the REM statement is merely a placekeeper for now, allowing us to refer to line 170 in line 290.

If the ball hasn't gone off the bottom edge of the screen, it must be erased so that it can be moved. Line 250 erases the ball from its old location by eXclusive ORing it off the screen. Line 250 then uses the POINT function to see whether there is anything on the screen where the center of the ball in its new location is to go. We have already checked whether the ball has hit the edge of the playing field or the paddle and moved it away if it has, and we have erased the ball temporarily. Therefore, if the POINT function reveals any color but black, the background color, then the ball must be hitting a brick. Because the brick is surrounded by black, it is simple to PAINT it black and thus erase it. The ball then reverses vertical direction as if it had struck the top or bottom edge of the screen, so that the brick seems to vanish and the ball to rebound from the collision.

On line 260 the ball is redrawn in its new location. Note that if the ball were not erased before the collision check, or if it were redrawn before the check, the entire brick collision sequence would not work, because the PAINT would erase the ball as well as the brick, and then the PUT statement with the XOR option would not function properly.

Finally, the program loops to line 190 to execute again the whole sequence of ball and paddle movements.

When you run the program in Listing 11-4, remember that it can only be terminated with Ctrl-Break.

Listing 11-4. Blockbuster—Ball.

```

10 REM Blockbuster--playfield, bricks, paddle, and ball.
100 DEFINT A-Z:SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
110 CIRCLE(3,3),2,3:PAINT STEP(0,0),3
120 DIM BALL(10):GET(0,0)-(5,5),BALL
130 LINE(0,0)-(319,199),2,BF:LINE(80,20)-(241,185),0,BF
140 FOR I=0 TO 7:FOR J=0 TO 3:LINE(82+20*I,48+J*12)
    -STEP(18,8),((I+J) MOD 2)+1,BF:NEXT J:NEXT I
160 PX=150:PXINC=0
170 REM *placekeeping line*
180 BX=80:BY=100:BXINC=4:BYINC=4:PUT(BX,BY),BALL
190 A$=INKEY$:IF A$="c" THEN PXINC=5 ELSE IF A$="z" THEN
    PXINC=-5 ELSE IF A$="x" THEN PXINC=0
200 OLDPX=PX:PX=PX+PXINC:IF PX<80 OR PX>221 THEN PX=OLDPX
210 LINE(OLDPX,181)-(OLDPX+20,181),0:
    LINE(PX,181)-(PX+20,181),3:OLDPX=PX
220 OLDBX=BX:BX=BX+BXINC:IF BX<80 OR BX>234 THEN
    BXINC=-BXINC:BX=BX+2*BXINC
230 OLDBY=BY:BY=BY+BYINC:IF BY<24 THEN BYINC=-BYINC:
    BY=BY+2*BYINC
240 IF BY>175 THEN IF BX<PX-5 OR BX>PX+20 THEN 280 ELSE
    BYINC=-BYINC:BY=BY+2*BYINC:BXINC=(BX-PX)\2-4
250 PUT(OLDBX,OLDBY),BALL:IF POINT(BX+2,BY+2)=0 THEN GOTO
    260 ELSE PAINT(BX+2,BY+2),0:BYINC=-BYINC:BY=BY+2*BYINC
260 PUT(BX,BY),BALL
270 GOTO 190
280 REM *placekeeping line*
290 PUT(OLDBX,OLDBY),BALL:GOTO 170

```

Programming the Housekeeping

Only a few details remain before *Blockbuster* is a working game, as shown in Listing 11-5. The program must check to see if any bricks remain and count the balls as they're lost. Line 150 sets the initial number of balls and bricks. Line 280 reduces the number of balls after each ball is lost and then loops to 170, which indicates on the screen how many balls remain and terminates the game with the message YOU LOST!!!! if no balls remain.

On line 250 the number of bricks is reduced each time a

brick is hit. Each time a brick is hit, line 250 also indicates on the screen how many bricks remain. If none remain, the game is ended with the message YOU WON!!!!.

The program in Listing 11-5 is *Blockbuster* in its entirety. Run the program, then review the listing and try to relate each line to what you see as you play. You might also find it worthwhile to follow the program through a few loops by modifying the code to print variables such as BXINC, BYINC, BX, and BY on an unused portion of the screen or on your printer as the game is playing.

Listing 11-5. Blockbuster—Finished Version.

```

10 REM Blockbuster---finished version.
100 DEFINT A-Z:SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
110 CIRCLE(3,3),2,3:PAINT STEP(0,0),3
120 DIM BALL(10):GET(0,0)-(5,5),BALL
130 LINE(0,0)-(319,199),2,BF:LINE(80,20)-(241,185),0,BF
140 FOR I=0 TO 7:FOR J=0 TO 3:LINE(82+20*I,48+J*12)
    -STEP(18,8),((I+J) MOD 2)+1,BF:NEXT J:NEXT I
150 NBALLS=3:NBRKS=32
160 PX=150:PXINC=0
170 LOCATE 2,2:PRINT "Balls left ";NBALLS:IF NBALLS=0 THEN
    LOCATE 15,13:PRINT "YOU LOST!!!!":GOTO 300
180 BX=80:BY=100:BXINC=4:BYINC=4:PUT(BX,BY),BALL
190 A$=INKEY$:IF A$="c" THEN PXINC=5 ELSE IF A$="z" THEN
    PXINC=-5 ELSE IF A$="x" THEN PXINC=0
200 OLDPX=PX:PX=PX+PXINC:IF PX<80 OR PX>221 THEN PX=OLDPX
210 LINE(OLDPX,181)-(OLDPX+20,181),0:
    LINE(PX,181)-(PX+20,181),3:OLDPX=PX
220 OLDBX=BX:BX=BX+BXINC:IF BX<80 OR BX>234 THEN
    BXINC=-BXINC:BX=BX+2*BXINC
230 OLDBY=BY:BY=BY+BYINC:IF BY<24 THEN BYINC=-BYINC:
    BY=BY+2*BYINC
240 IF BY>175 THEN IF BX<PX-5 OR BX>PX+20 THEN 280 ELSE
    BYINC=-BYINC:BY=BY+2*BYINC:BXINC=(BX-PX)\2-4
250 PUT(OLDBX,OLDBY),BALL:IF POINT(BX+2,BY+2)=0 THEN GOTO 260
    ELSE PAINT(BX+2,BY+2),0:BYINC=-BYINC:BY=BY+2*BYINC:
    NBRKS=NBRKS-1:LOCATE 2,20:PRINT "Bricks left";NBRKS:
    IF NBRKS=0 THEN LOCATE 15,13:PRINT "YOU WON!!!!":GOTO 300
260 PUT(BX,BY),BALL
270 GOTO 190
280 NBALLS=NBALLS-1
290 PUT(OLDBX,OLDBY),BALL:GOTO 170
300 LOCATE 25,9:PRINT "PRESS ANY KEY TO CONTINUE";
310 A$=INKEY$:IF A$="" THEN 310 ELSE CLS
320 END

```


THE BALL'S IN YOUR COURT

And that's all there is to programming a video game! *Blockbuster* isn't as polished as it might be; we left many rough edges for you to smooth out. The best way to acquire game-making skills is to experiment with the program and redesign the game. In this, the version of the game with comments, shown in Listing 11-6, will be of use.

Listing 11-6. Blockbuster—Finished, Commented Version.

```

10 REM Blockbuster--a pong-type video game--complete, commented
  version.
100 DEFINT A-Z:SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
105 REM Draw the ball and fill it in
110 CIRCLE(3,3),2,3:PAINT STEP(0,0),3
115 REM Save the ball in array called BALL
120 DIM BALL(10):GET(0,0)-(5,5),BALL
125 REM Draw the playfield
130 LINE(0,0)-(319,199),2,BF:LINE(80,20)-(241,185),0,BF
135 REM Draw the bricks in the playfield
140 FOR I=0 TO 7:FOR J=0 TO 3:LINE(82+20*I,48+J*12)-
  STEP(18,8),((I+J) MOD 2)+1,BF:NEXT J:NEXT I
145 REM Initialize the number of balls and number of bricks
150 NBALLS=3:NBRKS=32
155 REM Set initial paddle position and movement
160 PX=150:PXINC=0
165 REM Keep track of number of balls left, if none then end game
170 LOCATE 2,2:PRINT "Balls left ";NBALLS:IF NBALLS=0 THEN
  LOCATE 15,13:PRINT "YOU LOST!!!!":GOTO 300
175 REM Start new ball and direction
180 BX=80:BY=100:BXINC=4:BYINC=4:PUT(BX,BY),BALL
185 REM Set paddle direction according to what key is pressed
190 A$=INKEY$:IF A$="c" THEN PXINC=5 ELSE IF A$="z" THEN
  PXINC=-5 ELSE IF A$="x" THEN PXINC=0
195 REM Find new paddle location--do not let it go off playing
  field
200 OLDPX=PX:PX=PX+PXINC:IF PX<80 OR PX>221 THEN PX=OLDPX
202 REM Draw the paddle at its new location which then
  becomes the
204 REM old location
210 LINE(OLDPX,181)-(OLDPX+20,181),0:
  LINE(PX,181)-(PX+20,181),3:OLDPX=PX
215 REM Find new horizontal ball location--if it would hit a wall
  then go in other direction
220 OLDBX=BX:BX=BX+BXINC:IF BX<80 OR BX>234 THEN
  BXINC=-BXINC:BX=BX+2*BXINC
225 REM Find new vertical ball location--if it hits the top then
  go down

```

Listing 11-6—cont. Blockbuster—Finished, Commented Version.

```

230 OLDBY=BY:BY=BY+BYINC:IF BY<24 THEN BYINC=-BYINC:BY=BY+2*BYINC
232 REM If ball is at paddle row and missed the paddle then goto
    line 280
234 REM If ball is at paddle line and hit the paddle then set
    ball to go up
236 REM and set the horizontal movement according to where the
    paddle was
238 REM hit
240 IF BY>175 THEN IF BX<PX-5 OR BX>PX+20 THEN 280 ELSE
    BYINC=-BYINC:BY=BY+2*BYINC:BXINC=(BX-PX)\2-4
242 REM Erase the ball, restoring the background. If the dot at
    the center
244 REM of where the ball is to be put next is black then
    goto 260. If
246 REM not, then a brick has been hit, so erase the brick by
    PAINTing it
248 REM the background color, then reverse the direction of the
    ball and
249 REM reduce the brick count by one. If no bricks left, you
    won!!!
250 PUT(OLDBX,OLDBY),BALL:IF POINT(BX+2,BY+2)=0 THEN GOTO 260
    ELSE PAINT(BX+2,BY+2),0:BYINC=-BYINC:BY=BY+2*BYINC:
    NBRKS=NBRKS-1:LOCATE 2,20:PRINT "Bricks left";NBRKS:
    IF NBRKS=0 THEN LOCATE 15,13:PRINT "YOU WON!!!!":GOTO 300
255 REM Draw the ball at its new location
260 PUT(BX,BY),BALL
265 REM Loop back and do it all again
270 GOTO 190
275 REM The ball went past the paddle, so there is one less ball
280 NBALLS=NBALLS-1
285 REM Erase the old ball and goto where a new ball will be
    started
290 PUT(OLDBX,OLDBY),BALL:GOTO 170
295 REM Common proper exit for both winning and loosing
300 LOCATE 25,9:PRINT "PRESS ANY KEY TO CONTINUE";
310 A$=INKEY$:IF A$="" THEN 310 ELSE CLS
320 END

```

The most obvious omission in *Blockbuster* is the lack of sound. BASIC provides an impressive array of noise-creating commands. BEEP is the simplest, producing a single short tone. SOUND produces a specified tone of specified duration. PLAY is the aural equivalent of the DRAW statement, a small language in its own right. Experiment with

adding these to *Blockbuster*. Remember, though, that the longer the sound, the slower the game action.

Many of the details of any game are what is commonly called "human engineering." These are elements which have nothing to do with the play of the game *per se*, but which are intended to reduce the player's frustration. An example of this is the way we started the game with the ball aimed directly at the paddle, so the ball wouldn't be lost immediately when a beginner was playing. Similarly, it would be a good idea to pause after a ball is lost to allow the player to recover. You could design the program so the player would press a certain key to start the ball, thus ensuring that the player always would be prepared. This would give him a feeling of control, and might make the game pace more pleasant, as there are currently no breaks in the flow of the game.

Another nice touch would be a key that halted the action until it was pressed again. Such a key allows acknowledgment of a ringing phone or the "call of nature." While Ctrl-Num Lock accomplishes the same result, it can be hard to press two keys at once in the midst of concentrating on a game, so a single key is preferable.

Both the start and end of the game can have added features. At the player's option, the game could provide instructions when it is run initially, and could be started at the player's leisure with the press of a key. Upon conclusion, the winning or losing message (which might be better placed) can be displayed for several seconds, then the instruction screen redisplayed, so the game can be restarted or ended with the press of a key. These additional features make it easier for the player to play multiple games; players are never in a situation where they can't figure out what to do next. The program should provide a self-contained, self-explanatory environment for the player.

In addition to the preceding features, multiple screens and skill levels can be added to *Blockbuster*. There are many possible brick layouts of varying degrees of difficulty. Rather than ending when the bricks are cleared, the

game could progress to more difficult screens, so the player's increasing skill could be matched. Similarly, the speed could start out slow and increase, so players would be constantly challenged. (Incidentally, it would be possible to compress the program code into fewer lines so it would run faster. We have spread the code over many lines only for explicative purposes.) The start point of the ball could also be varied using the RND function, so the game would be less predictable.

One other possible enhancement would be the maintenance of a high score on disk. It is truly interesting how hard people will play a game for the honor of having their score displayed.

No doubt, you will find other shortcomings of the current version of *Blockbuster*. By all means, fix them! You cannot simply read your way to becoming a game designer. You must design and program. Before proceeding to the next chapter, dissect *Blockbuster* and make at least one change to the program. Only in this way will you progress.

CHAPTER 12

THE DRAW STATEMENT—A LANGUAGE WITHIN A LANGUAGE

For all the power of the graphics commands you've learned so far, these commands can be awkward to use and often require many program lines for relatively simple applications. Once again, BASIC provides an alternative in the form of the DRAW statement. The **DRAW** statement is a small graphics language within the BASIC language, complete with a set of one-letter commands. Draw can do anything that the PSET or LINE statements can and usually can do so a good deal more easily.

The DRAW statement is best visualized as a set of commands controlling a pen which draws on the screen. (When we use “pen” to describe what DRAW does, we mean an imaginary pen that draws pixels on the screen.) The pen can make linear moves to any point on the screen, with the moves described in either absolute or relative coordinates. The pen can leave a trail in any available color, or it can be “lifted” from the screen as it moves. Figures can be scaled and rotated 90 degrees. We will begin our discussion of the DRAW statement by covering basic movement around the screen.

MOVING AROUND THE SCREEN—THE M SUBCOMMAND

First we will explain how the DRAW statement works. Its form is **DRAW commandstring** where the characters in

the string **commandstring** are commands to the DRAW command. For example, type:

NEW and press Enter

to clear memory and reset all DRAW parameters to their default state. Clear the screen in medium-resolution mode, and type:

DRAW "M100,100M199,199" and press Enter

The string following DRAW instructs BASIC to move to location (100,100) and then (199,199), leaving a line as it goes. The parameter commandstring may be either a string constant, as above, or a string variable, as in **A\$="M100,100M199,199": DRAW A\$**.

A summary of subcommands to the DRAW statement is shown in Table 12-1.

The simplest DRAW subcommand is the move subcommand, M, which is used as **DRAW "Mx,y"** where the letter M is the move subcommand, and x and y are the new coordinates. Both x and y are required.

It is important to understand that the M subcommand is not like PSET in that it draws a dot at the specified location. A trace of all movement is left on the screen unless explicitly turned off. (More on lifting the pen later.) For example, clear the screen and type:

PSET (50,50): DRAW "M150,150" and press Enter

The PSET statement sets the last point referenced to (50,50). The M subcommand to the DRAW statement then moves the pen from the last point referenced to (150, 150), drawing as it goes. To emphasize—DRAW leaves pixels everywhere it goes, not just where it arrives.

X and y may be specified in either absolute or relative coordinates, but relative coordinates are specified differently than with other graphics commands. If x is prefixed with a plus sign (+) or a minus sign (-), then x and y are treated as relative coordinates. Relative coordinates are taken relative to the last point referenced. DRAW sets the last point referenced to the last pen position, even if the pen did not draw at that position. The last point referenced is especially important to the DRAW statement

Table 12-1. DRAW Statement Subcommands

DRAW SUBCOMMAND	DESCRIPTION
Mx,y	Move. If x has a plus or minus sign, the move is relative to the last point referenced; otherwise, the move is absolute to (x,y). If the move is relative, distances represented by x and y are determined by the S subcommand.
Un	Move up. The distance moved is n units; actual distance covered is determined by the S subcommand.
Dn	Move down.
Rn	Move right.
Ln	Move left.
En	Move up and right (northeast).
Fn	Move down and right (southeast).
Gn	Move down and left (southwest).
Hn	Move up and left (northwest).
B	Blank (prefix). Don't draw any points when executing the next subcommand.
N	No move (prefix). Return to starting position after executing the next subcommand.
An	Angle. Rotate figures 0, 90, 180, or 270 degrees, with n equal to 0, 1, 2, or 3, respectively.
Cn	Color. Select color 0, 1, 2, or 3 to draw with.
Sn	Scale. Select figure scaling size, with 1 the smallest and 255 the largest. The default is n=4, where 1 unit=1 pixel.
Xn\$;	Execute substring. The specified DRAW command string is executed, and then the current command string continues. With the X subcommand, one DRAW command string can execute one or more other command strings.

because a line is drawn between the last point referenced and the new point.

As an example of screen addressing, clear the screen and type:

```
PSET (100,100): DRAW "M150,100": DRAW "M+0,50":  
DRAW "M-50,-50" and press Enter
```

The first DRAW specifies the address in absolute fashion, while the last two DRAWs use relative addressing.

Multiple DRAW subcommands can be combined in a single string, thus making the DRAW statement extremely compact. The preceding example could just as well have been written as **PSET (100,100): DRAW "M150,100 M+0,50 M-50,-50"**. (The spaces are optional.) DRAW commands can be a little cryptic at times, but then again consider that it would have taken three LINE statements to produce the same result, and we have barely tapped the DRAW subcommands. There is, however, a separator that can be used to improve readability if necessary. Semicolons can be inserted between subcommands, and, in fact, are required in certain circumstances. The previous example is certainly clearer as **PSET (100,100): DRAW "M150,100; M+0,50; M-50,-50"** than as one long string, and semicolons become more useful as command string lengths increase.

BASIC strings can be as long as 255 characters. A DRAW command string of this length would be difficult to decipher, even with semicolons. One DRAW command picks up where the last left off, however, so that long command strings usually are not needed; several short (and more readable) strings almost always will do as well.

There is one quirk of the M subcommand. The DRAW statement allows the use of a scaling factor, so the distance drawn by DRAW subcommands is dependent on the current scaling factor. For example, if the scale factor is 1/2 normal, then each DRAW subcommand will move only 1/2 as far across the screen as it would normally. This can be useful in that the same DRAW command string can draw a figure in any number of sizes, as the situation requires, simply by changing the scaling factor. This scale factor, which we will discuss in detail later, affects relative addressing as used with the M subcommand. For example, if the scale factor is 1/2 normal, then **DRAW "M+20,20"** will only move to a point 10 pixels below and 10 pixels to the right of the last point referenced. However, the M subcommand using absolute addressing is not affected by the scaling factor.

The default for the scale factor is 1 distance unit equals 1 pixel, so, until the scale factor is changed, distance is measured in pixels. Until we reach the scale subcommand, we will treat distance and number of pixels as synonymous, so you need not worry about the scale factor.

MOVING AROUND THE SCREEN—THE MOVEMENT SUBCOMMANDS

In many cases, there is a shorter way to accomplish what the M subcommand does when it uses relative addressing. There are four subcommands: U, D, L, and R, that move the fictional screen pen up, down, left, and right, respectively, a specified distance. The U subcommand is **DRAW "Un"** where **n** is the distance to move. If **n** is omitted, it defaults to 1. As discussed above, due to the scale factor, distance is not necessarily the same as number of pixels, but we will treat the two as the same for now. The other three subcommands take the same form, except, of course, that the appropriate letter D, L, or R is substituted for U.

For example, clear the screen and type:

```
DRAW "U40 L60 D40 R60" and press Enter
```

to draw a rectangle. Each subcommand starts from the end point of the last subcommand. Note that this is considerably more compact and intelligible than **DRAW "M+0,-40 M-60,0 M+0,40 M+60,0"**.

There is the question, where the first DRAW begins if no last point referenced exists. Fortunately, this point is well defined—it is (160,100). In fact, each time the screen is cleared, the last point referenced for DRAW is reset to (160,100).

There are four more movement subcommands: E, F, G, and H. These commands move diagonally up and right, down and right, down and left, and up and left, respectively. It is easy to remember these subcommand if you can remember that the E subcommand points up and

right, or northeast. Then the other three subcommands fall into place as we move clockwise, as shown in Fig. 12-1.

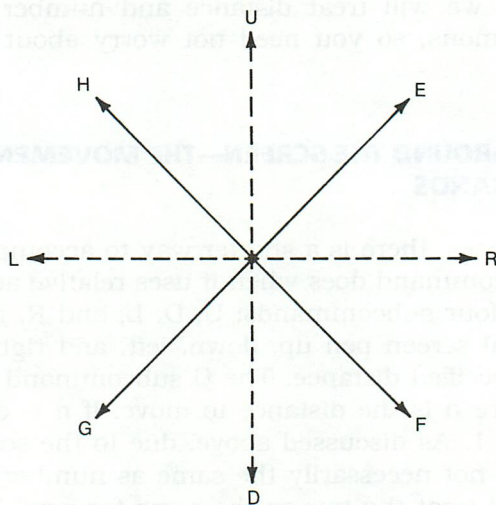


Fig. 12-1. Action of DRAW Movement Subcommands.

The E subcommand takes the form **DRAW "En"** and the others are similar, just as with the first four movement subcommands. The distance represented by **n** is somewhat different with this set of four subcommands, because pixels are spaced farther apart in a diagonal direction than they are horizontally or vertically. Thus, a given distance, as specified by **n**, will be about 1.4 times as far diagonally as horizontally or vertically. This effect makes it easier to join lines in a command such as **DRAW "R10 D10 H10"**, so you probably will find it quite useful; just bear in mind that this effect does occur. If **n** is omitted, it defaults to 1.

The use of these four movement subcommands is straightforward. For example, clear the screen and type:

DRAW "E30 F50 G30 H50" and press Enter

All eight movement subcommands, along with the M subcommand, can be used together. The program shown in Listing 12-1 uses the M subcommand to make an initial

absolute move, and thereafter uses the movement subcommands. The M subcommand is preferable to the movement subcommands only when either an absolute move is desired or a move that is not diagonal, horizontal, or vertical (e.g., "M+10,20") is to be made.

Listing 12-1. DRAW Statement Movement Subcommands.

```

100 REM Program to demonstrate DRAW statement movement
110 REM   subcommands by drawing a kite.
120 SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
130 DRAW "m240,40"           'Draw string
140 DRAW "u20 d40 u20 l20 r40" 'Draw cross
150 DRAW "g20 h20 e20 f20"    'Draw outline
160 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
170 A$=INKEY$:IF A$="" THEN 170 ELSE CLS
180 END

```

CONTROLLING THE PEN

So far, we can move wherever we want on the screen, but we always leave a line behind as we move. As we control the imaginary pen, we sometimes want to be able to lift it off the screen as we move it, leaving no line. This is accomplished with the B subcommand.

The B, or blank, subcommand operates as a prefix to any of the subcommands that set screen position. If a subcommand is prefixed with B, the action the subcommand causes has no effect on the screen. For example, **DRAW "BM+99,99"** draws no line, but moves the last point referenced, as evidenced by typing (without clearing the screen):

DRAW "'U100'" and press Enter

The B subcommand only lifts the pen for the duration of the single subcommand following it, so it is never necessary to explicitly reset DRAW to draw on the screen. To keep the pen off the screen for several successive subcommands, each must be separately prefixed with B.

There is another pen control subcommand, N, which simplifies drawing complex figures. Like the B sub-

command, the N subcommand operates as a prefix to any subcommand that sets screen position. Any subcommand prefixed by N will execute its action normally; upon finishing, N-prefixed subcommands will reset the last point referenced to the same point as when the subcommand began. This makes it easy to draw figures around a central point. For example, clear the screen and type:

`DRAW "NU30 NR30 ND30 NL30"` and press Enter

Note that each subcommand starts from the same point, even though each terminates at a different point. The N subcommand makes DRAW command strings more compact and more readable, and makes errors less likely than if the programmer had to retrace his steps to get back to the original point. (While there is no obvious mnemonic for the N subcommand, it can perhaps be thought of as the "no-move" subcommand.)

THE COLOR SUBCOMMAND

There are three subcommands that set the environment in which other DRAW subcommands execute. The first of these subcommands is the C, or **color**, subcommand.

The C subcommand selects the color in which the line is to be drawn. It is used as `DRAW "Cn"` where **n** is 0 for the background color, or 1-3 for the colors of the selected palette in medium-resolution graphic mode. All pixels affected by the DRAW statement are drawn in color **n** until another C subcommand is executed. If **n** is out of range, color 3 will be used.

For example, clear the screen and type:

`DRAW 'C1 L50 C3 D40 C2 M+50,-40'` and press Enter

to use each of the three palette colors. The background color can be used to erase. For example, leave the screen unchanged and type:

`DRAW 'C0 L50 D40 M+50,-40'` and press Enter

to erase the image we just drew. We have drawn the figure

again in color 0. Because color 0 is the color the background is drawn in, redrawing the figure in color 0 is the same as erasing the figure.

THE SCALE SUBCOMMAND

The S, or scale, subcommand sets the number of pixels represented by the distance parameter to DRAW subcommands. Varying the scale factor causes the size of any figure drawn with the DRAW statement to vary proportionately.

The S subcommand is used as **DRAW "Sn"** where **n** is the scale factor, in the range 1 to 255. An out-of-range value for **n** will result in an "Illegal function call" error. The scale factor operates by setting the number of pixels drawn for each unit of distance to $n/4$. For example, if **n** is 4, then $n/4=1$ pixel is drawn for each unit of distance specified. Clear the screen and type:

DRAW "S4 R10" and press Enter

to set the scale factor to 4 and draw 10 distance units = 10 pixels to the right. 4 is the default scale factor.

Similarly, leave the screen unchanged and type:

DRAW "S1 BU8 L40" and press Enter

This only produces a line the length of the first line, even though the distance value is 40, because 40 is scaled to 10 by the $1/4$ scale factor.

Scale factor enables us to draw a figure easily in any desired size. For example, store a diamond shape in **D\$** by typing:

D\$="E20 F20 G20 H20" and press Enter

Now clear the screen and draw the figure in four sizes by typing:

DRAW "BM0,100 S1 "+D\$ and press Enter

DRAW "BM20,100 S4 "+D\$ and press Enter

DRAW "BM70,100 S8 "+D\$ and press Enter

DRAW "BM160,100 S16 "+D\$ and press Enter

(Use the edit keys to reuse the same line at the top of the screen for the DRAW statements, so the text doesn't wander into the forms we've drawn.) Clearly, it is no trouble at all to draw a form in any size using DRAW, and this is as true for complex forms such as schematic symbols as for a simple diamond.

The preceding example may require a little explanation. We initially store the DRAW command string for a diamond in the string variable D\$. We then set the last point referenced, without drawing, with BM, set the scale factor with S, and draw the diamond with D\$. The plus sign (+), when used with strings, concatenates them; that is, the plus sign joins the two strings into one string. In the first DRAW statement in the preceding example, the DRAW "BM0,100 S1 "+D\$ is precisely the same as DRAW "BM0,100 S1 E20 F20 G20 H20". Using string variables and combining strings in this fashion greatly enhances the power of the DRAW statement.

THE ANGLE SUBCOMMAND

Another DRAW subcommand is A, the angle subcommand. This subcommand rotates a figure by 0, 90, 180, or 270 degrees. What happens is that the A subcommand rotates the reference axes for purposes of the DRAW statement. Because all DRAW subcommands work with reference to the new axes, the effect is to "spin" the entire figure so it is aligned with the new axes. This means that if the new angle is, for example, 90 degrees, U (up) becomes L (left), L becomes D (down), and so on. Unfortunately, this effect is limited somewhat more than we might wish because the reference axes can only be turned multiples of 90 degrees.

The A subcommand is DRAW "An" where n is 0, 1, 2, or 3, corresponding to rotations of 0, 90, 180, or 270 degrees. An out-of-range value for n will result in an "Illegal function call" error. The absolute addressing version of the M subcommand is not affected by the A subcommand.

As an example of the A subcommand, define a flat rec-

tangle with **F\$="Ü1Ø R6Ø D1Ø L6Ø"**, then clear the screen and type:

DRAW "AØ"+F\$ and press Enter

This is the unrotated rectangle and is equivalent to **DRAW F\$** in the default (unrotated) case.

Now clear the screen and type:

DRAW "A1"+F\$ and press Enter

and you have a tall rectangle. To see the full range of rotation, leave the screen unchanged and type:

DRAW "A2"+F\$: DRAW "A3"+F\$: DRAW "AØ"+F\$ and press Enter

Note that the angle set with the A subcommand remains in effect until changed. To return to the normal reference axes, **DRAW "AØ"** must be typed.

Within the limits imposed by the 90 degree minimum rotation, the A subcommand is an easy way to rotate a figure from its normal attitude. Changing the angle is not particularly useful if you are going to draw a figure only once, but it can be useful if a predefined figure is to be shown in a number of attitudes. For example, define the image of an elongated X with **X\$="F3Ø NE15 NF1Ø NG15"**. Now type:

CLS: DRAW "BM1Ø,65 AØ"+X\$: DRAW "BM7Ø,1ØØ A1"+X\$
and press Enter
DRAW "BM18Ø,1ØØ A2"+X\$: DRAW "BM26Ø,65 A3"+X\$
and press Enter

The figure is drawn so it appears to be caught in several stages of spinning. Here, the A subcommand has saved having to specify four images, instead allowing us to use one image, contained in X\$, and four rotations.

The same approach can be extended to figures as complicated as a cartwheeling human; often four rotations will serve admirably to create the impression of motion, particularly in conjunction with animation via the PUT graphics statement. In such a case, the image is defined and drawn first in all four rotations with the A subcommand. The

rotations are saved to an array with the GET statement and are available then to be drawn rapidly with the PUT statement as discussed in Chapter 10. The program shown in Listing 12-2 illustrates this concept.

Listing 12-2. DRAW Statement Animation.

```

100 REM Program to demonstrate animation based on rotations
110 REM with the DRAW statement.
120 SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
130 REM Define image to be rotated
140 IMAGE$="bm+6,-5 dl0 l12 ul0 r4 br4 r4 bl6 bd3 m+0,0"
150 REM Draw and GET image rotation #0
160 DRAW "c3 a0 bml0,l0"+IMAGE$
170 DIM ANGLE0(100):GET(0,0)-(20,20),ANGLE0
180 REM Draw and GET image rotation #1
190 DRAW "c3 a1 bm30,l0"+IMAGE$
200 DIM ANGLE1(100):GET(20,0)-(40,20),ANGLE1
210 REM Draw and GET image rotation #2
220 DRAW "c3 a2 bm50,l0"+IMAGE$
230 DIM ANGLE2(100):GET(40,0)-(60,20),ANGLE2
240 REM Draw and GET image rotation #3
250 DRAW "c3 a3 bm70,l0"+IMAGE$
260 DIM ANGLE3(100):GET(60,0)-(80,20),ANGLE3
270 REM Draw across screen in alternating rotations.
280 REM Rotations #1 and #3 are transposed in sequence
290 REM so that the rotation is in the correct direction
300 FOR I=0 TO 280 STEP 8
310 PUT(I,90),ANGLE0:FOR J=1 TO 80:NEXT J:PUT(I,90),ANGLE0
320 PUT(I+2,90),ANGLE3:FOR J=1 TO 80:NEXT J:PUT(I+2,90),ANGLE3
330 PUT(I+4,90),ANGLE2:FOR J=1 TO 80:NEXT J:PUT(I+4,90),ANGLE2
340 PUT(I+6,90),ANGLE1:FOR J=1 TO 80:NEXT J:PUT(I+6,90),ANGLE1
350 NEXT I
360 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
370 A$=INKEY$:IF A$="" THEN 370 ELSE CLS
380 END

```

The A subcommand is useful especially for manipulation of drawings of component parts, such as resistors or capacitors. Whole schematics can be produced with a few appropriately located and rotated DRAW command strings. Likewise, a floor plan could be produced with command strings for objects such as tables and chairs, rotated as needed.

There is one other item regarding the A subcommand. As you may remember from our discussion of aspect ratio in Chapter 7, there is some confusion about the proper

correction for the screen aspect. The A subcommand has the property of correcting for screen aspect when rotating to 90 and 270 degrees (4/3 ratio), so that any figure will appear to be approximately the same dimensions whether drawn horizontally or vertically.

BASIC does, in fact, use 5/6 to correct for screen aspect here as well. For example, type:

```
T$="U50 R60 D50 L60" and press Enter
```

and then clear the screen and type:

```
DRAW "A0"+T$: DRAW "A1"+T$: DRAW "A2"+T$: DRAW  
"A3"+T$ and press Enter
```

and see that each square is of exactly the same dimension. Presto, BASIC has corrected a 5/6 aspect ratio form, which appears to be a square, to the proper dimensions so it continues to appear to be a square when rotated on its side.

VARIABLES IN THE DRAW STATEMENT

All the command strings we have described have drawn what are essentially fixed figures. For example, while we can draw any given rectangle, we have no way to draw a rectangle of variable dimensions. The DRAW statement does, however, allow us to use variables as easily as constants.

Clear the screen and type:

```
FOR I=10 TO 50 STEP 10: DRAW "R=I;D=I;L=I;U=I;":  
NEXT I and press Enter
```

Notice that this effect is not accomplished with the S subcommand; instead, we are specifying distances with variables rather than with constants.

Variables to the DRAW statement are indicated by preceding them with an equal sign (=) and following them with a semicolon (;) in the form **DRAW "z=n;"** where **z** is any of the above DRAW subcommands, and **n** is the variable parameter. In this case, the semicolon is required. Note that any number in any DRAW subcommand may be

replaced with a variable, including either x or y in the M subcommand, and the parameters to the C, A, and S subcommands. In fact, array variables of any dimension may be used. For example, in **DIM A(10): A(2)=20: I=2: DRAW "U=A(I);"**. This gives enormous flexibility to the DRAW statement, which, with variable parameters, becomes capable of producing virtually any display, such as the concentric diamonds produced by the program shown in Listing 12-3. The program is more compact than any that could be written with the PSET or LINE statements. In combination with the scaling and angle capabilities of the DRAW statement, variables make the DRAW statement the fastest and most compact way to execute graphics from BASIC on the IBM PC. Yet, there is still one more element to the DRAW statement.

Listing 12-3. Variable Parameters To DRAW.

```

100 REM Program to demonstrate variable parameters to DRAW.
110 SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
120 REM Set string to draw diamond
130 DIAMOND$="bu8 f8 g8 h8 e8 bd8"
140 DRAW "bm160,100"      'Set center point
150 FOR SCALE=2 TO 36 STEP 2      'Draw 18 diamonds
160   DRAW "s=scale;"+DIAMOND$    'Draw scaled diamond
170 NEXT SCALE
180 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
190 A$=INKEY$:IF A$="" THEN 190 ELSE CLS
200 END

```

DRAW SUBSTRINGS

The preferred method of programming, especially for complex programs, is to write subroutines that execute basic functions, and then to use these subroutines as building blocks to construct the whole program. This approach has the virtue of allowing the task to be broken into smaller portions, which are designed and debugged more easily, and which can be used readily by several sections of the program or, indeed, by several programs. A similar approach is possible with the DRAW statement via the X subcommand.

The X subcommand executes a DRAW command string as a substring. The X subcommand is **DRAW "Xstring;"** where **string** is a valid DRAW command string, and the semicolon is required. For example, clear the screen and type:

```
A$="U10 R10 G10": DRAW "M100,100 X A$;" and press
Enter
```

(The spaces are optional.) This produces results identical to **A\$="U10 R10 G10": DRAW "M100,100 "+A\$**, so why is the X subcommand particularly useful? Well, remember that BASIC strings are limited to 255 characters, so that strings could only be combined so long as the overall length was less than 255. With the X subcommand, DRAW command strings can be of any length. Also, because the X subcommand is specifically designed to execute substrings, it is easier to interpret when trying to determine just what a DRAW command string does.

The program shown in Listing 12-4 DRAWS by executing substrings. Type and run it, noting the compactness and clarity of the program.

If the substring contains an invalid command string, an "Illegal function call" error will result.

Variable parameters in the form "**=n;**" are not permitted with the X subcommand. However, there is, another way to accomplish the same effect. A string variable can contain the name of the substring variable to be executed. While this concept is none too straightforward, it adds to the flexibility of the DRAW statement. For example, clear the screen and type:

```
A$="U20 R20":B$="D20 L20" and press Enter
```

to define two different substrings. Now type:

```
C$="A$": DRAW "X "+C$+";" and press Enter
```

to execute substring A\$, the name of which is assigned to C\$, and then type:

```
C$="B$": DRAW "X "+C$+";" and press Enter
```

to set the contents of C\$ to the string "B\$", then to exe-

Listing 12-4. DRAW Statement Substring Commands.

```

100 REM Program to demonstrate use of substring subcommands
110 REM with the DRAW statement.
120 SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
130 REM Set substring to draw a square
140 S$="r24 d20 124 u20"
150 DRAW "c3" 'Set color to white
160 REM Draw some scaled squares
170 DRAW "bm10,10" 'Upper left corner
180 FOR I=1 TO 16 STEP 2 '8 sizes
190 DRAW "s=i; x s$;"
200 NEXT I
210 DRAW "s4" 'Reset scale
220 REM Animate the square
230 PRESET(10,140):DRAW "x s$;" 'Draw initial square
240 FOR I=10 TO 260 STEP 10 'Move square 26 times
250 DRAW "bm=i;,140 c0 x s$;" 'Erase old square
260 IAHEAD=I+10 'Point to new square
270 DRAW "bm=iahead;,140 c3 x s$;" 'Draw new square
280 NEXT I
290 REM Rotate the square
300 DRAW "bm240,50" 'Common corner of squares
310 FOR I=0 TO 3 'Draw all 4 rotations
320 DRAW "a=i; x s$;" 'Draw the rotated square
330 NEXT I
340 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
350 A$=INKEY$:IF A$="" THEN 350 ELSE CLS
360 END

```

cute substring B\$. (Remember that the contents of a string variable are no different from a string constant.)

If you find this last section confusing, don't worry; this is an advanced technique which you can manage without. However, it does expand the potential power and ease of use of the DRAW statement.

The X subcommand is useful for constructing a figure by drawing one section at a time via substrings. For example, a building could be drawn by repeatedly using a few substrings that hold the commands to draw various windows and doors.

The X subcommand is also useful for compacting program code and for making DRAW command strings more readable.

ERROR CONDITIONS

There are few circumstances in which the DRAW statement will produce an error message. It does not matter if coordinates are off the screen; DRAW will plot the pixels at the nearest on-screen point and maintain the off-screen coordinate as the last point referenced. This is similar to what happens to off-screen points with the PSET statement, except that a point is always plotted with the DRAW statement. In practical terms, this means that if your program DRAWS off the screen, a line along the border of the screen will be produced, a line that will consist of the nearest on-screen point for all the off-screen points plotted.

The only error message produced by DRAW is “Illegal function call”. Potential causes include out-of-range parameters to the A or S subcommands (though not to the C subcommand), characters that are not subcommands, and the use of invalid strings as parameters to the X subcommand.

SUMMARY OF THE DRAW STATEMENT

The DRAW statement is really a small graphics-oriented language. The subcommands are only one letter long, and the effects are confined solely to graphics, but DRAW subcommands are complete, powerful, and self-contained. Just as the other graphics capabilities of BASIC are an advance on crude text-mode graphics, DRAW is an advance over the other BASIC graphics statements taken as a whole. DRAW is one of the features that makes the IBM PC an exceptional graphics computer.

The DRAW statement operates faster in drawing a complex figure than LINE and PSET do. Even so, speed is a limitation. The graphics GET and PUT statements, however, can be used in conjunction with the DRAW statement to produce the rapid graphics required for animation.

As your graphics projects become more ambitious, the DRAW statement will become increasingly important. You will discover that you have developed a set of “primi-

tives"—DRAW command strings that define frequently used building-block forms—you will use in many programs. Also, as your programs become larger, space considerations will become important, and there is no more compact way to execute graphics than with the DRAW statement. Whenever you consider how to approach a graphics problem, think of the DRAW statement; few are the applications that cannot be most quickly and efficiently accomplished with DRAW.

A CHARACTER-GENERATION PACKAGE

In earlier chapters, we described the limitations of printing text in graphics mode. BASIC can print characters only every 8 pixels, and can print characters only of a certain size and shape. Nor does BASIC provide us with an easy way to produce italics, boldface, underlining, subscripts, superscripts, or many foreign and scientific characters. However, the PC is fully capable of displaying virtually any characters desired.

In this chapter, we will demonstrate a character-generation package, consisting of two subroutines, that allows flexible, user-definable character generation and which can be inserted into and used from any BASIC program. The key to this package is the power of the DRAW statement.

Fig. 11 in the color photograph section gives you an idea of what this character-generation package can do. In this chapter, we will first define the nature of text, then explore the character-generation package, and finally describe the workings of the sample program and how you can integrate the package into your own BASIC programs.

THE NATURE OF TEXT

Text consists of pixels organized to look like characters. For example, the letter A appears as shown in Fig. 13-1. This is true in both text and graphics modes. When we use

PRINT in graphics mode, BASIC draws the pattern of the specified characters for us. In this case, we are limited to only one pattern for each character BASIC knows and to only the locations at which BASIC can PRINT. Alternatively, we could draw the characters, then be able to draw patterns in any size and at any location that we desire. This would seem to require an overwhelming array of graphics statements and program code. However, DRAW lets us reduce the entire character-generation process to less than 10 program lines, plus one line for each character we wish to generate!

To see how DRAW can be used to draw a character, clear the screen in medium-resolution mode and type:

```
A$="A0 BD1 D5 BU2 R4 BD2 U5 H1 L2 G1 BU1": DRAW
A$ and press Enter
```

and the letter A will appear. This character is a collection of lines, composed of pixels, and created with DRAW. It is possible to relate each of the DRAW subcommands to the form of the letter A. For instance, the D5 subcommand draws the left side of the letter.

The C, A, and S subcommands to DRAW make it easy to implement color, rotation, and scaling capabilities. For example, type:

```
DRAW "BR20 A2 S8"+A$ and press Enter
```

where A\$ is defined in our last example, to draw a large, upside-down A. Type:

```
DRAW "A0 S4" and press Enter
```

to restore the normal unrotated, unscaled DRAW environment.

DRAW provides all the tools we require to implement character generation—the challenge is to make these capabilities readily available to the BASIC programmer. We must make it possible for a BASIC program to specify the color, angle, and size of the text to be printed and to print the text not only in the normal fashion—horizontally from left to right—but with any desired direction and spacing the programmer desires.

SUBROUTINES

A subroutine is a section of a BASIC program that can be executed from another part of the program by the statement **GOSUB n** where **n** is the line number of the first line of the subroutine. The subroutine ends with a **RETURN** statement, and the program resumes execution at the statement after the **GOSUB** that called the subroutine.

The advantage of a subroutine is that it can be called from many places in a program, yet the subroutine exists in only one place. That is, **GOSUB** statements that call the subroutine can appear in several locations in the program, but the subroutine will only appear once, saving space and programming time. Also, the lines that constitute the subroutine can be transferred to another program intact, so the subroutine need only be developed once but can be used over and over again.

A subroutine is perfect for our character generator. Our subroutine can be located at high line numbers where one normally does not place program code, then it can be put into any program and called with the same **GOSUB** in every case. Our subroutine is located at lines 60000-60290, so it can always be "called" by the statement **GOSUB 60000** in any program that includes the subroutine. The user will need to define the string to be printed, set any other characteristics (such as color) that he wishes to change, and then call the subroutine.

We also need a second subroutine that will be called once, at the beginning of the program, to initialize the character-generation subroutine.

STRUCTURE OF THE CHARACTER GENERATOR

We will specify a **DRAW** command string that draws each character. We will also define several parameters that can be set by the user to specify color, rotation, scaling, and horizontal and vertical distance between letters. Then, when the user requests that a string be put on the screen, the program will find the **DRAW** command string that corresponds to the first character in the string, and execute it

with the specified color, rotation, and scaling. It will do the same for the second character, and so on, until each character in the string is drawn on the screen.

The process of selecting the proper DRAW command string for the specified character is simple. We will establish a string which contains each of the characters we might want to generate. For example, the first character would be A, the second B, and so on. We will call this the *reference string*. We will also establish a string array. The first element of the string array will be the DRAW command string for our first character, A. The second element will be the DRAW command string for the second character, B, and so on for all the characters to be generated. We will call this the *character array*. We have only to determine the location in the reference string of any character to be generated, and then the corresponding DRAW command string will be selected from the array and executed to place the character on the screen.

We need two subroutines: the first to initialize the parameters, reference string, and character array; and the second to execute the DRAW command string for each of the characters to be generated.

THE INITIALIZATION SUBROUTINE

The parameters and building blocks for character generation must be initialized before the characters can be drawn. The program code for the initialization subroutine is shown in Listing 13-1. (Do not enter this listing—it is not a complete program and will not run.) This is a subroutine that must be called once, at the beginning of the program, by executing the statement **GOSUB 60300**. Line 60370 sets default values for the character-generation parameters. The rotation angle, ZZA, is set to zero, specifying no rotation. The color, ZZC, is set to three, the color white. The scale factor, ZZS, is set to four, so that figures are scaled to normal size. The horizontal distance between the upper-left corners of the characters, ZZX, is seven, and the vertical distance between the characters, ZZY, is zero. This will produce white, normal-sized characters from left

to right across the screen as is the case with text created with PRINT.

Line 60370 also sets aside room for the character array, which will contain the DRAW command strings that define the characters. We will only deal with the 26 letters of the alphabet, plus the always useful space character, so the string array ZZA\$, which will hold the DRAW command strings, has 27 storage locations.

Notice that we start all variables associated with character generation with the characters ZZ. This is because we want to make sure that normal program variables, such as I, J, COUNT, and MAXX, do not have the same names. Using the same name for two different purposes would cause confusion and unpredictable program behavior. Few people start their variable names with ZZ.

Lines 60390-60670 define the DRAW command strings for the characters. Each of lines 60410 through 60670 stores the DRAW string for one letter of the alphabet in the character array, ZZA\$. Thus, line 60410 stores the DRAW command string for A, line 60420 stores the string for B, and line 60660 stores the string for Z. Line 60670 stores a null string for the space character. Note that comments on the right of each line indicate which character the line defines.

Line 60390 sets up the reference string ZZZ\$, which is used to find the command string corresponding to the character to be printed. For example, the first character in ZZZ\$ is A, and the first array element in the character array ZZA\$ is the DRAW command string for the character A. Likewise, the twenty-sixth character in ZZZ\$ is Z, and ZZA\$(26) is the string that draws the character Z. Once lines 60390-60670 are executed, the appropriate DRAW command string for any character in ZZZ\$ can be selected based on that character's location in ZZA\$.

When the initialization subroutine is finished, the default parameters are set, and the arrays ZZA\$ is set to draw any character contained in ZZ\$. To add more characters, more lines after 60670 would be added, each of which would store the DRAW command string for a given character in ZZA\$. The character array ZZA\$ would have

to be enlarged to hold the new strings. The characters then would be added to the reference string ZZZ\$ in positions corresponding to their locations in array ZZA\$.

Listing 13-1. Initializing Character-Generation Package I.

```

60300 REM *****
60310 REM Set reference and character strings.
60320 REM This subroutine initializes the character
60330 REM generation package—it should be called
60340 REM just once before the first call to the
60350 REM character drawing subroutine.
60360 REM *****
60370 DIM ZZA$(27):ZZA=0:ZZC=3:ZZS=4:ZZX=7:ZZY=0
60380 REM ZZZ$ is a reference string used to find the print char
60390 ZZZ$="ABCDEFGHIJKLMNOPQRSTUVWXYZ "
60400 REM Each of the strings below is used to define
60410 ZZA$(1)="BD1D5BU2R4BD2U5HLL2G1BU1" 'A
60420 ZZA$(2)="D6R3E1U1H1L3BR3E1U1H1L3" 'B
60430 ZZA$(3)="BD1D4F1R2E1BU4H1L2G1BU" 'C
60440 ZZA$(4)="D6R3E1U4H1L3" 'D
60450 ZZA$(5)="D6R4BU3BL1L3BU3R4BL4" 'E
60460 ZZA$(6)="D6BU3R3BL3U3R4BL4" 'F
60470 ZZA$(7)="BD1D4F1R3U3L1BU3BR1L3G1BU1" 'G
60480 ZZA$(8)="D6BU3R4BD3U6BL4" 'H
60490 ZZA$(9)="BD6BR1R2BL1U6BR1L2BL1" 'I
60500 ZZA$(10)="BD5F1R1E1U5BL3" 'J
60510 ZZA$(11)="D6BU3R1F3BU6G3L1BU3" 'K
60520 ZZA$(12)="D6R4BL4BU6" 'L
60530 ZZA$(13)="D6BR4U6G2H2" 'M
60540 ZZA$(14)="D6BR4U6BD4H4" 'N
60550 ZZA$(15)="BD1D4F1R2E1U4H1L2G1BU1" 'O
60560 ZZA$(16)="D6BU3R3E1U1H1L3" 'P
60570 ZZA$(17)="BD1D4F1R1BR2H2BF1BG1E2U3H1L2G1BU1" 'R
60580 ZZA$(18)="D6BR4H3BL1R3E1U1H1L3" 'S
60590 ZZA$(19)="BD6R3E1U1H1L2H1U1E1R3BL4" 'T
60600 ZZA$(20)="BD6BR2U6BR2L4" 'U
60610 ZZA$(21)="D5F1R3E1U5BL4" 'V
60620 ZZA$(22)="D3F1D1F1E1U1E1U3BL4" 'W
60630 ZZA$(23)="D5F1E1U1BD1F1E1U5BL4" 'X
60640 ZZA$(24)="BD6U1E4U1BD6U1H4U1" 'Y
60650 ZZA$(25)="D2F2D2BU2E2U2BL4" 'Z
60660 ZZA$(26)="BD6BR4L4U1E4U1L4" '<SPACE>
60670 ZZA$(27)=""
60680 RETURN

```

A note about designing characters: we have not described the design of characters in detail. Type any of

the DRAW command strings on lines 60410-60670 to verify that they work; you can check each of the DRAW subcommands in any string with a pencil and paper to see *how* they work. Usually, graph paper is the best way to design characters. Draw the character by hand first, with each box on the graph paper representing one pixel, then translate the dot pattern into a DRAW command string.

The DRAW command string must always end at the same point it started (the upper-left corner of the character), so the program can space properly to the next character. The B subcommand can be used to move back to the starting point without drawing, if necessary.

There is one problem to be aware of when designing characters. If a character is drawn as a group of dots, the spacing between dots will increase when we scale the character larger. It is essential, therefore, to construct the character only out of lines, and these lines must meet at all corners.

For example, clear the screen and type:

```
A$="BD1 D4 F1 R3 U3 L1 BU3 BR1 L3 BL1": DRAW A$
and press Enter
```

This defines and draws the character G, as shown in Fig. 13-2. Note that the left side and top bars of the character are never joined. With normal scaling, this doesn't matter, because the ends are adjacent. However, type:

```
DRAW "BR1Ø S16"+A$ and press Enter
```

to draw a larger character. The gap becomes obvious. This character is more properly defined as `A$="BD1 D4 F1 R3 U3 L1 BU3 BR1 L3 G1 BU1"` as shown in Fig. 13-3. Here, the second to last subcommand G1 connects the two sides of the character. Type the preceding command to redefine A\$ and type:

```
DRAW "BD1Ø"+A$ and press Enter
```

and you will see that there is no longer any gap.

THE CHARACTER-DRAWING SUBROUTINE

The actual drawing of the characters is done by the subroutine at lines 60000-60290 as shown in Listing 13-2. (Do

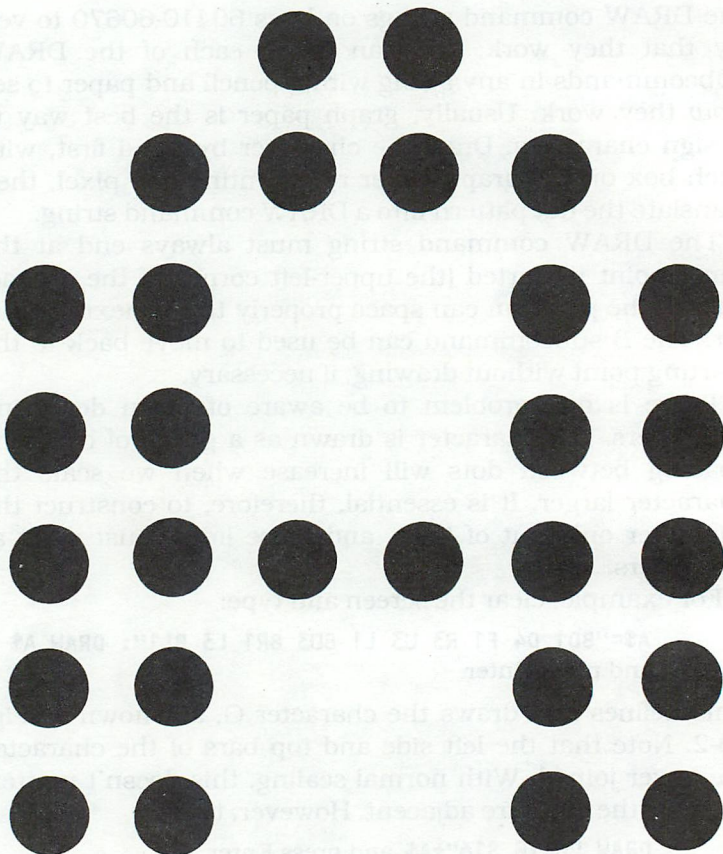


Fig. 13-1. Pixels in letter "A".

not enter this listing.) Note that most of this subroutine consists of REM statements, and only seven lines are required to do all the drawing. The many REM statements ensure that it will be easy for you and others to use the character-generator package in other programs.

The subroutine functions like a new BASIC statement, complete with parameters. These parameters are several BASIC variables which "give instructions" to the subroutine. The parameters are described on lines 60070-60160. Here, ZZS contains the scaling factor, ZZA selects the rotation angle, and ZYC selects the color in which the

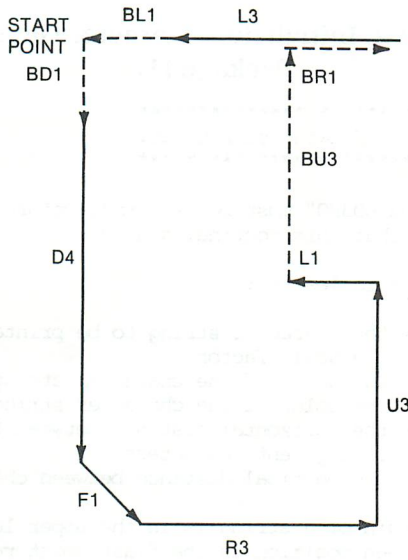


Fig. 13-2. Letter "G" with gap.

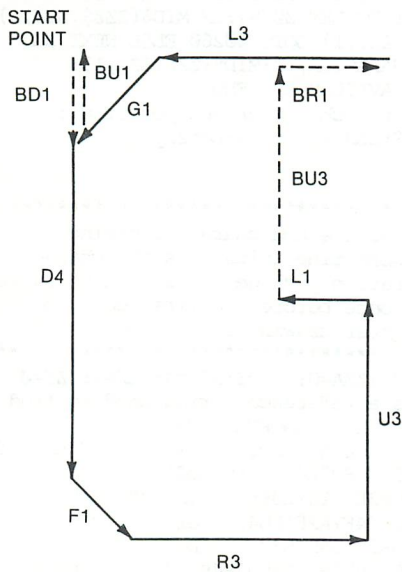


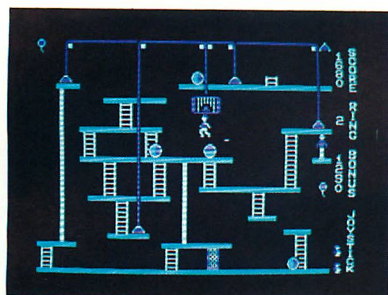
Fig. 13-3. Letter "G" without gap.

Listing 13-2. Initializing Character-Generation Package II.

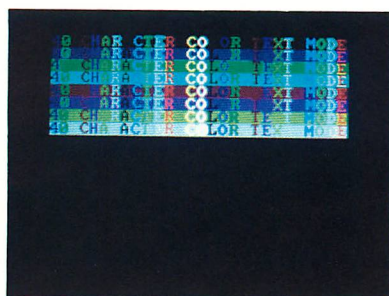
```

60000 REM *****
60010 REM Character drawing subroutine.
60020 REM *****
60030 REM
60040 REM A "GOSUB 60300" must be performed before the first
60050 REM time that this routine is used
60060 REM
60070 REM Input variables are:
60080 REM
60090 REM   zz$ - the character string to be printed
60100 REM   zzs - the scale factor
60110 REM   zza - the angle of the character string
60120 REM   zzc - the color of the character string
60130 REM   zzx - the horizontal distance between the origins
60140 REM           of adjacent characters
60150 REM   zzy - the vertical distance between characters
60160 REM
60170 REM The origin of a string is in the upper left corner
60180 REM The screen position is the "last point referenced"
60190 REM and can be set easily with a PRESET(x,y)
60200 REM
60210 DRAW "s=zzs;c=zzc;a=zza;" 'Set scale,color and angle
60220 FOR ZZI=1 TO LEN(ZZ$) ' for each character
60230 REM Locate the character in the reference string
60240 FOR ZZJ=1 TO LEN(ZZZ$):IF MID$(ZZZ$,ZZJ,1)=
MID$(ZZ$,ZZI,1) GOTO 60260 ELSE NEXT ZZJ
60250 PRINT "CHARACTER, ";MID$(ZZ$,ZZI,1);
" IS NOT AVAILABLE ":END
60260 REM Draw the character and position for next character
60270 DRAW ZZA$(ZZJ)+"br=zzx;bd=zzy;"
60280 NEXT ZZI
60290 RETURN
60300 REM *****
60310 REM Set reference and character strings.
60320 REM This subroutine initializes the character
60330 REM generation package--it should be called
60340 REM just once before the first call to the
60350 REM character drawing subroutine.
60360 REM *****
60370 DIM ZZA$(27):ZZA=0:ZZC=3:ZZS=4:ZZX=7:ZZY=0
60380 REM ZZZ$ is a reference string used to find the print char
60390 ZZZ$="ABCDEFGHJKLMNOPQRSTUVWXYZ "
60400 REM Each of the strings below is used to define
60410 ZZA$(1)="BD1D5BU2R4BD2U5H1L2G1BU1" 'A
60420 ZZA$(2)="D6R3E1U1H1L3BR3E1U1H1L3" 'B
60430 ZZA$(3)="BD1D4F1R2E1BU4H1L2G1BU" 'C
60440 ZZA$(4)="D6R3E1U4H1L3" 'D
60450 ZZA$(5)="D6R4BU3BL1L3BU3R4BL4" 'E
60460 ZZA$(6)="D6BU3R3BL3U3R4BL4" 'F

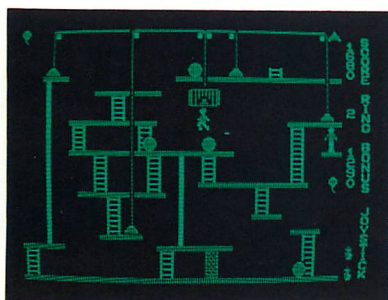
```

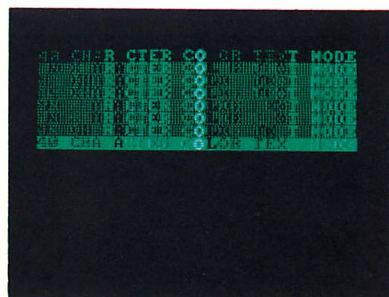
1A RGB monitor.



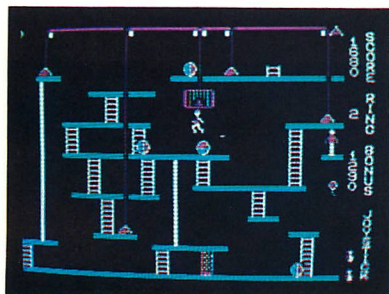
2A RGB monitor screen quality.



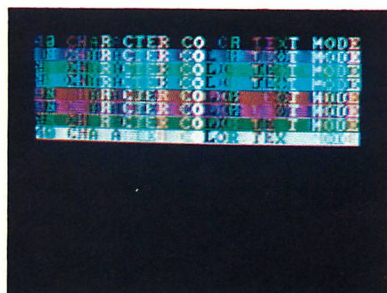
1B Black and green composite monitor.



2B Black and green composite monitor screen quality.



1C Color television.



2C Color television screen quality.

Fig. 1 Examples of typical graphics displays.

Fig. 2 Sample run of Color Text Modes program.

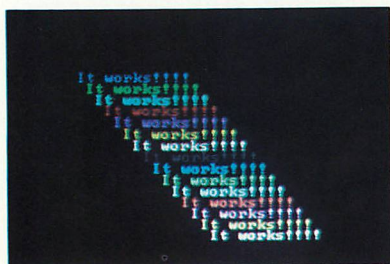


Fig. 3 Sample run of Graphics Display Test program.

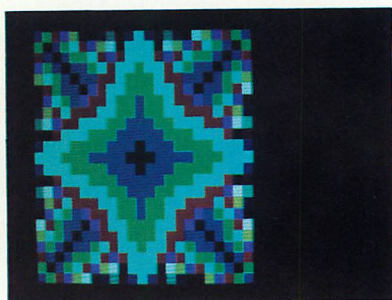
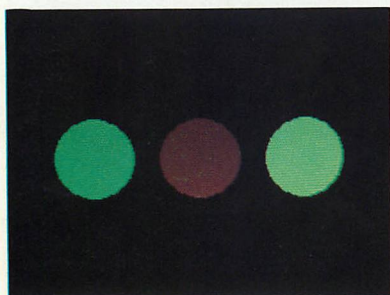
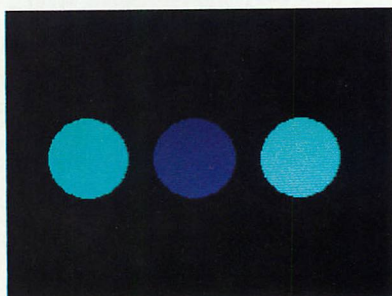


Fig. 4 Sample run of Kaleidoscope program.



5A Palette 0.



5B Palette 1.

Fig. 5 Sample run of Palette 0 and 1 Demonstration program.

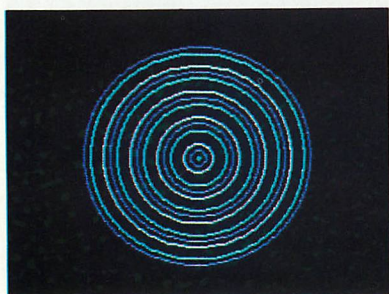


Fig. 6 Sample run of CIRCLE Statement program.

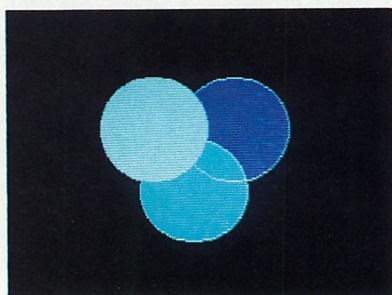


Fig. 7 Sample run of PAINT Statement program.

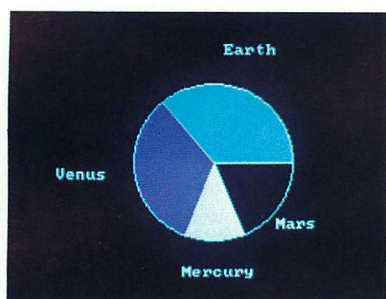


Fig. 8 Sample run of Pie Chart program.

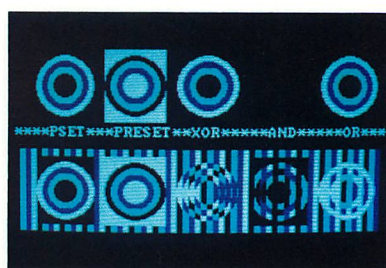


Fig. 9 Sample run of PUT Statement Options program.

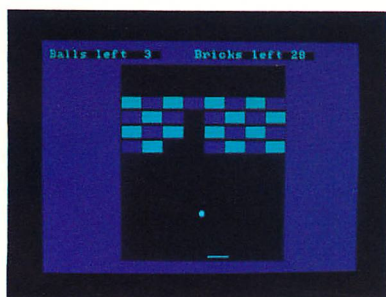


Fig. 10 Sample run of Blockbuster—Finished Version program.

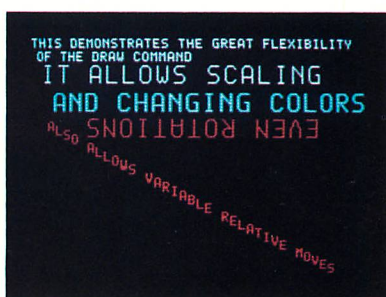


Fig. 11 Sample run of Character Generation Package program.

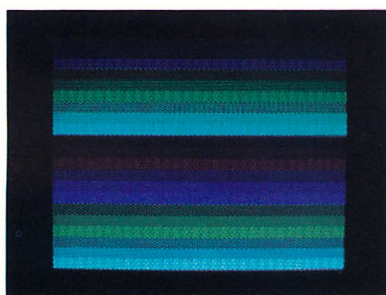


Fig. 12 Sample run of Text Mode Color Shading program.

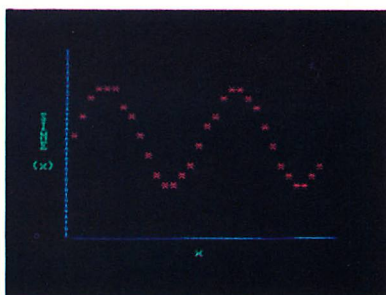
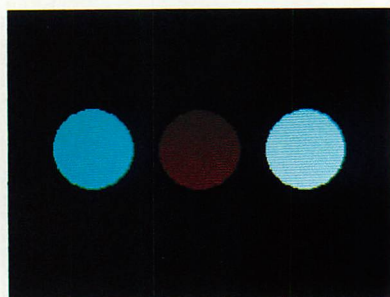


Fig. 13 Sample run of Text-Mode Graphics program.



**Fig. 14 Sample run of
Color Generation by
Artfacting program.**



**Fig. 15 Sample run of
Three Color Palettes
program—Unofficial
Palette.**

Listing 13-2—cont. Initializing Character-Generation Package II .

```

60470 ZZA$(7)="BD1D4F1R3U3L1BU3BR1L3G1BU1" 'G
60480 ZZA$(8)="D6BU3R4BD3U6BL4" 'H
60490 ZZA$(9)="BD6BR1R2BL1U6BR1L2BL1" 'I
60500 ZZA$(10)="BD5F1R1E1U5BL3" 'J
60510 ZZA$(11)="D6BU3R1F3BU6G3L1BU3" 'K
60520 ZZA$(12)="D6R4BL4BU6" 'L
60530 ZZA$(13)="D6BR4U6G2H2" 'M
60540 ZZA$(14)="D6BR4U6BD4H4" 'N
60550 ZZA$(15)="BD1D4F1R2E1U4H1L2G1BU1" 'O
60560 ZZA$(16)="D6BU3R3E1U1H1L3" 'P
60570 ZZA$(17)="BD1D4F1R1BR2H2BF1BG1E2U3H1L2G1BU1"
60580 ZZA$(18)="D6BR4H3BL1R3E1U1H1L3" 'R
60590 ZZA$(19)="BD6R3E1U1H1L2H1U1E1R3BL4" 'S
60600 ZZA$(20)="BD6BR2U6BR2L4" 'T
60610 ZZA$(21)="D5F1R3E1U5BL4" 'U
60620 ZZA$(22)="D3F1D1F1E1U1E1U3BL4" 'V
60630 ZZA$(23)="D5F1E1U1BD1F1E1U5BL4" 'W
60640 ZZA$(24)="BD6U1E4U1BD6U1H4U1" 'X
60650 ZZA$(25)="D2F2D2BU2E2U2BL4" 'Y
60660 ZZA$(26)="BD6BR4L4U1E4U1L4" 'Z
60670 ZZA$(27)="" '<SPACE>
60680 RETURN

```

characters are to be drawn. The variables **ZZX** and **ZZY** indicate the horizontal and vertical spacing between characters. These parameters are set in the initialization subroutine to produce standard text—white, of normal size, and printed horizontally from left to right. To change any of these characteristics, the program assigns a new value to the corresponding parameter. For example, executing **ZZC=2** before calling the subroutine with **GOSUB 60000** would cause the text to be colored magenta. These parameters retain their values if not explicitly changed. For example, if the above example were executed, all characters would be drawn in magenta until **ZZC** was changed again.

The one required parameter to the subroutine is the string containing the characters to be put on the screen. This string, which we will call the *print string*, is named **ZZ\$**. Whenever a **GOSUB 60000** is executed, the character-drawing subroutine attempts to draw whatever

is contained in ZZ\$. We will now examine the subroutine to see how this works.

Line 60210 is the first line executed. This line sets the scaling factor, color, and rotation angle based on the values of ZZS, ZZC, and ZZA, respectively. There's nothing tricky here, just a standard DRAW command string using variables.

The FOR . . . NEXT loop across lines 60220 to 60280 draws each character in the print string in turn. The function LEN(ZZ\$) returns the length of the print string ZZ\$, so the values of ZZI, the FOR . . . NEXT variable, are 1, 2, 3, and so on up to the length of ZZ\$. The value of ZZI is used to select each of the characters in the print string in turn.

The inner FOR . . . NEXT loop on line 60240 runs through each character in turn in the string ZZZ\$, the reference string that we defined in the initialization subroutine. The MID\$ function picks out a substring from a string. The first MID\$ function on line 60240, in conjunction with the inner loop, picks out each character in turn from the reference string. The second MID\$ function picks out the current character (as specified by ZZI) from the print string. In effect, the current character from the print string is compared to each of the reference characters, until a match is found or until all the characters in the reference string have been checked.

If no match is found, the character to be printed is not one the program is designed to produce. For example, we have not defined any strings for numbers. In this case, line 60250 informs the user, through a statement on the screen, that the character is unavailable, and then terminates the program.

If a match is found in reference string ZZZ\$, line 60270 executes the DRAW command string (selected from the character array ZZA\$) that corresponds to that character. For example, if the character B is to be drawn, a match will be made between the print string and reference string when ZZJ is 2, and the DRAW command string ZZA\$(2) will be executed, producing the character B. The second part of the DRAW statement on line 60270 moves the last

point referenced (the point at which the next character will begin to be drawn) over by the distance specified by ZZX and ZZY. In the default case, where ZZX is 7 and ZZY is 0, this means that the next character will be placed to the right on the same line.

All that really occurs on line 60270 is that the character to be printed is compared to each character in the reference string ZZZ\$ to determine which DRAW command string stored in the character array ZZA\$ should be executed to draw the proper character. Each character in the print string ZZ\$ is drawn in this manner, as the outer FOR . . . NEXT loop selects each of the characters in the print string in turn.

USING THE CHARACTER-GENERATION PACKAGE

The character-generation package can be inserted into any program in the form shown in Listing 13-2. The only limitations are that the program can have no other lines in the range 60000-60680, and the program should have no variable names starting with the letters ZZ.

Before any characters can be generated, the package must be initialized with the statement **GOSUB 60300**. Thereafter, all that is needed to generate characters is to set the last point referenced to the desired starting point for the text, set any parameters we wish to change, store the text to be printed in ZZ\$, and execute a **GOSUB 60000**. For example, the program in Listing 13-3 is a simple test of the package, merely generating characters under the default conditions.

The real power of the package lies in the optional parameters ZZS, ZZA, ZZC, ZZX, and ZZY. The program shown in Listing 13-4 demonstrates the use of each of these parameters as shown in Fig. 11 in the color photograph section. Enter and run it now.

Note that this sample program requires less than 20 lines, apart from the subroutines, to put many forms of text on the screen. Now that we have completed the character-generation package, text in many colors, locations, rotations, and sizes is easy to create.

Listing 13-3. Character Generation Package Test.

```

100 REM Simple program to demonstrate character
110 REM generation package.
120 REM Set medium resolution mode
130 SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
140 REM Initialize the character generation subroutine
150 GOSUB 60300
160 REM Print test message
170 ZZ$=" IF YOU CAN READ THIS THEN IT WORKS"
180 PRESET(10,30):GOSUB 60000
190 LOCATE 24,6:PRINT "PRESS ANY KEY TO CONTINUE";
200 A$=INKEY$:IF A$="" THEN 200 ELSE CLS
210 END
60000 REM *****
60010 REM Character drawing subroutine.
60020 REM *****
60030 REM
60040 REM A "GOSUB 60300" must be performed before the first
60050 REM time that this routine is used
60060 REM
60070 REM Input variables are:
60080 REM
60090 REM zz$ - the character string to be printed
60100 REM zzs - the scale factor
60110 REM zza - the angle of the character string
60120 REM zzc - the color of the character string
60130 REM zzx - the horizontal distance between the origins
60140 REM of adjacent characters
60150 REM zzy - the vertical distance between characters
60160 REM
60170 REM The origin of a string is in the upper left corner
60180 REM The screen position is the "last point referenced"
60190 REM and can be set easily with a PRESET(x,y)
60200 REM
60210 DRAW "s=zzs;c=zzc;a=zza;" 'Set scale,color and angle
60220 FOR ZZI=1 TO LEN(ZZ$) ' for each character
60230 REM Locate the character in the reference string
60240 FOR ZZJ=1 TO LEN(ZZZ$):IF MID$(ZZZ$,ZZJ,1)=
MID$(ZZ$,ZZI,1) GOTO 60260 ELSE NEXT ZZJ
60250 PRINT "CHARACTER, ";MID$(ZZ$,ZZI,1);
" IS NOT AVAILABLE ":END
60260 REM Draw the character and position for next character
60270 DRAW ZZA$(ZZJ)+"br=zzx;bd=zzy;"
60280 NEXT ZZI
60290 RETURN
60300 REM *****
60310 REM Set reference and character strings.
60320 REM This subroutine initializes the character
60330 REM generation package--it should be called
60340 REM just once before the first call to the
60350 REM character drawing subroutine.
60360 REM *****

```


Listing 13-3—cont. Character Generation Package Test.

```

60370 DIM ZZA$(27):ZZA=0:ZZC=3:ZZS=4:ZZX=7:ZZY=0
60380 REM ZZZ$ is a reference string used to find the print char
60390 ZZZ$="ABCDEFGHIJKLMNOPQRSTUVWXYZ "
60400 REM Each of the strings below is used to define
60410 ZZA$(1)="BD1D5BU2R4BD2U5H1L2G1BU1" 'A
60420 ZZA$(2)="D6R3E1U1H1L3BR3E1U1H1L3" 'B
60430 ZZA$(3)="BD1D4F1R2E1BU4H1L2G1BU" 'C
60440 ZZA$(4)="D6R3E1U4H1L3" 'D
60450 ZZA$(5)="D6R4BU3BL1L3BU3R4BL4" 'E
60460 ZZA$(6)="D6BU3R3BL3U3R4BL4" 'F
60470 ZZA$(7)="BD1D4F1R3U3L1BU3BR1L3G1BU1" 'G
60480 ZZA$(8)="D6BU3R4BD3U6BL4" 'H
60490 ZZA$(9)="BD6BR1R2BL1U6BR1L2BL1" 'I
60500 ZZA$(10)="BD5F1R1E1U5BL3" 'J
60510 ZZA$(11)="D6BU3R1F3BU6G3L1BU3" 'K
60520 ZZA$(12)="D6R4BL4BU6" 'L
60530 ZZA$(13)="D6BR4U6G2H2" 'M
60540 ZZA$(14)="D6BR4U6BD4H4" 'N
60550 ZZA$(15)="BD1D4F1R2E1U4H1L2G1BU1" 'O
60560 ZZA$(16)="D6BU3R3E1U1H1L3" 'P
60570 ZZA$(17)="BD1D4F1R1BR2H2BF1BG1E2U3H1L2G1BU1"
60580 ZZA$(18)="D6BR4H3BL1R3E1U1H1L3" 'R
60590 ZZA$(19)="BD6R3E1U1H1L2H1U1E1R3BL4" 'S
60600 ZZA$(20)="BD6BR2U6BR2L4" 'T
60610 ZZA$(21)="D5F1R3E1U5BL4" 'U
60620 ZZA$(22)="D3F1D1F1E1U1E1U3BL4" 'V
60630 ZZA$(23)="D5F1E1U1BD1F1E1U5BL4" 'W
60640 ZZA$(24)="BD6U1E4U1BD6U1H4U1" 'X
60650 ZZA$(25)="D2F2D2BU2E2U2BL4" 'Y
60660 ZZA$(26)="BD6BR4L4U1E4U1L4" 'Z
60670 ZZA$(27)="" '<SPACE>
60680 RETURN

```

The sample program initializes the character-generation package on line 150.

Line 160 defines the first print string, "THIS DEMONSTRATES THE GREAT FLEXIBILITY". Line 170 uses the PRESET statement to set the point at which the text will begin to (10,5). (The M subcommand to DRAW, prefixed with the B subcommand, also could have been used.) Line 170 then calls the character-drawing subroutine to put the specified text on the screen, starting at the last point referenced, which is (10,5). The text is white, unrotated, and of normal size, because the default values of parameters ZYC, ZZA, and ZYS have not been changed.

Listing 13-4. Character-Generation Package.

```

100 REM Demo program for character generation subroutine.
110 REM
120 REM Set medium resolution mode
130 SCREEN 1,0:COLOR 0,1:KEY OFF:CLS
140 REM Initialize the character generation subroutine
150 GOSUB 60300
160 ZZ$="THIS DEMONSTRATES THE GREAT FLEXIBILITY"
170 PRESET(10,5):GOSUB 60000
180 ZZ$="OF THE DRAW COMMAND"
190 PRESET(15,15):GOSUB 60000
200 ZZ$="IT ALLOWS SCALING"
210 PRESET(20,25):ZZS=8:GOSUB 60000
220 ZZ$="AND CHANGING COLORS"
230 PRESET(30,45):ZZC=1:GOSUB 60000:PRESET(31,45):GOSUB 60000
240 ZZ$="EVEN ROTATIONS"
250 ZZC=2:PRESET(250,75):ZZA=2:GOSUB 60000:PRESET(251,75):
    GOSUB 60000
260 ZZ$="ALSO ALLOWS VARIABLE RELATIVE MOVES"
270 PRESET(25,65):ZZA=0:ZZS=4:ZZY=3:GOSUB 60000:PRESET(26,65):
    GOSUB 60000
280 LOCATE 24,8:PRINT "PRESS ANY KEY TO CONTINUE";
290 A$=INKEY$:IF A$="" THEN 290 ELSE CLS
300 END
60000 REM *****
60010 REM Character drawing subroutine.
60020 REM *****
60030 REM
60040 REM A "GOSUB 60300" must be performed before the first
60050 REM   time that this routine is used
60060 REM
60070 REM Input variables are:
60080 REM
60090 REM   zz$ - the character string to be printed
60100 REM   zzs - the scale factor
60110 REM   zza - the angle of the character string
60120 REM   zzc - the color of the character string
60130 REM   zzx - the horizontal distance between the origins
60140 REM           of adjacent characters
60150 REM   zzy - the vertical distance between characters
60160 REM
60170 REM The origin of a string is in the upper left corner
60180 REM The screen position is the "last point referenced"
60190 REM   and can be set easily with a PRESET(x,y)
60200 REM
60210 DRAW "s=zzs;c=zzc;a=zza;" 'Set scale,color and angle
60220 FOR ZZI=1 TO LEN(ZZ$) ' for each character
60230   REM Locate the character in the reference string
60240   FOR ZZJ=1 TO LEN(ZZ$):IF MID$(ZZZ$,ZZJ,1)=
       MID$(ZZ$,ZZI,1) GOTO 60260 ELSE NEXT ZZJ
60250   PRINT "CHARACTER, ";MID$(ZZ$,ZZI,1);
       " IS NOT AVAILABLE ":END

```


Listing 13-4—cont. Character-Generation Package.

```

60260 REM Draw the character and position for next character
60270 DRAW ZZA$(ZZJ)+"br=zzx;bd=zy;"
60280 NEXT ZZI
60290 RETURN
60300 REM *****
60310 REM Set reference and character strings.
60320 REM This subroutine initializes the character
60330 REM generation package—it should be called
60340 REM just once before the first call to the
60350 REM character drawing subroutine.
60360 REM *****
60370 DIM ZZA$(27):ZZA=0:ZZC=3:ZZS=4:ZZX=7:ZZY=0
60380 REM ZZZ$ is a reference string used to find the print char
60390 ZZZ$="ABCDEFGHIJKLMNOPQRSTUVWXYZ "
60400 REM Each of the strings below is used to define
60410 ZZA$(1)="BD1D5BU2R4BD2U5H1L2G1BU1" 'A
60420 ZZA$(2)="D6R3E1U1H1L3BR3E1U1H1L3" 'B
60430 ZZA$(3)="BD1D4F1R2E1BU4H1L2G1BU1" 'C
60440 ZZA$(4)="D6R3E1U4H1L3" 'D
60450 ZZA$(5)="D6R4BU3BL1L3BU3R4BL4" 'E
60460 ZZA$(6)="D6BU3R3BL3U3R4BL4" 'F
60470 ZZA$(7)="BD1D4F1R3U3L1BU3BR1L3G1BU1" 'G
60480 ZZA$(8)="D6BU3R4BD3U6BL4" 'H
60490 ZZA$(9)="BD6BR1R2BL1U6BR1L2BL1" 'I
60500 ZZA$(10)="BD5F1R1E1U5BL3" 'J
60510 ZZA$(11)="D6BU3R1F3BU6G3L1BU3" 'K
60520 ZZA$(12)="D6R4BL4BU6" 'L
60530 ZZA$(13)="D6BR4U6G2H2" 'M
60540 ZZA$(14)="D6BR4U6BD4H4" 'N
60550 ZZA$(15)="BD1D4F1R2E1U4H1L2G1BU1" 'O
60560 ZZA$(16)="D6BU3R3E1U1H1L3" 'P
60570 ZZA$(17)="BD1D4F1R1BR2H2BF1BG1E2U3H1L2G1BU1"
60580 ZZA$(18)="D6BR4H3BL1R3E1U1H1L3" 'R
60590 ZZA$(19)="BD6R3E1U1H1L2H1U1E1R3BL4" 'S
60600 ZZA$(20)="BD6BR2U6BR2L4" 'T
60610 ZZA$(21)="D5F1R3E1U5BL4" 'U
60620 ZZA$(22)="D3F1D1F1E1U1E1U3BL4" 'V
60630 ZZA$(23)="D5F1E1U1BD1F1E1U5BL4" 'W
60640 ZZA$(24)="BD6U1E4U1BD6U1H4U1" 'X
60650 ZZA$(25)="D2F2D2BU2E2U2BL4" 'Y
60660 ZZA$(26)="BD6BR4L4U1E4U1L4" 'Z
60670 ZZA$(27)="" '<SPACE>
60680 RETURN

```

Lines 180 and 190 put additional text on the screen, with the same characteristics as the first text displayed. Here the PRESET statement sets the text starting location to (15,15) below and to the right of the first text.

Line 210 sets ZZS, the scaling factor, to 8. As you will recall from our description of DRAW, this means text will be double-sized. Indeed, the string "IT ALLOWS SCALING" is enlarged. A statement such as **ZZS=8** is all that is ever required to change the scaling factor. The new scaling factor remains in effect until changed.

Line 230 sets ZYC to 1. This means that text will be drawn in cyan. Note that we execute two GOSUB 60000 statements on line 230, the second offset by one column from the first. This is because single colored pixels do not show up well on many displays. The two executions of the character-drawing subroutine cause double-width characters to be drawn. Because we haven't changed ZZS since we set it to 8, these characters are enlarged.

Line 250 sets ZYC to 2, so that magenta text will be produced. ZZA is set to 2; all DRAW subcommands will operate at 180 degrees from normal. Thus, the text is drawn upside-down and right to left. Sideways text, reading bottom to top or top to bottom, would be as easy to produce with ZZA values of 1 or 3. Again, we execute the drawing routine twice, with an offset of one column, so that the colored text will show up well.

Finally, line 270 sets the angle back to unrotated, the scaling back to normal, and ZZY, the vertical offset between adjacent characters, to 3. This causes the text to descend as it prints from left to right, so that a slanting line of text is produced. PRESET, ZZA, ZZX, and ZZY can be combined to produce letters in any of four rotations anywhere on the screen and following one another in virtually any pattern.

INSERTING THE PACKAGE IN YOUR PROGRAMS

The easiest way to insert the character-generation package into another program is with the MERGE command. First, type both subroutines as shown in Listing 13-2. Then, save the program lines to disk with the command **SAVE "CHARGEN",A**. The trailing A is required. Then the package can be inserted into any program by simply typing **MERGE "CHARGEN"**.

For example, type:

NEW and press Enter

to clear memory, type the lines shown in Listing 13-2, and save the character-generation code by typing:

SAVE "CHARGEN",A and press Enter

Then type:

NEW and press Enter

to clear memory again; this simulates leaving BASIC and restarting it at a later time. Type only the lines in Listing 13-3 with numbers lower than 60000 and type:

MERGE "CHARGEN" and press Enter

The LIST command shows that the program is complete.

Note: If your program has any lines numbered 60000-60680 as in the character-generation package, they will be wiped out and replaced by the package when the MERGE command is executed.

SUMMARY OF THE CHARACTER-GENERATION PACKAGE

We have designed a character-generation package that overcomes the problems of the BASIC PRINT statement and which is flexible and easy to use. It is possible to design such a useful and compact package only because of the power of DRAW.

We have designed this package to be completely functional and ready to use. This does not mean that there are not enhancements you might make. The character set could be expanded to include lowercase and special characters. Other fonts, such as italic, gothic, and mathematical notation, could be designed. An ambitious but useful project would be to make the package capable of producing any of several fonts as selected by the main program.

The great virtue of this is that once it was designed and working properly, you would need only include the package in your graphing, text processing, and game programs to have any of a number of character types, sizes, colors, and rotations available anywhere on the screen.

HIGH-RESOLUTION GRAPHICS MODE

High-resolution graphics mode allows the finest detail possible on the IBM PC. In this mode, the screen consists of 640 columns by 200 rows of dots. Each of these dots, however, can be only black or white; this is the “trade-off” for the excellent detail available. Functionally, high-resolution mode is similar to medium-resolution mode, except that every inch across the screen contains twice as many dots, and there is no color. Because twice as many pixels are required to draw the same figure, high-resolution graphics mode commands are slower than their medium-resolution graphics mode counterparts.

High-resolution mode is ideal for detailed graphics of any kind, such as graphing, mapping, and games, as long as the application does not require color. (There are two distinct ways to produce some color in high-resolution graphics mode; neither of these is supported by BASIC, and neither is particularly easy to use. Color in high-resolution mode will be covered in Chapter 19.)

Because every graphics command available in medium-resolution graphics mode works similarly in high-resolution graphics mode, we will not cover these commands in detail here. Rather, we will describe only those features of each command unique to high-resolution mode and will provide an example of each command.

One point to be made is that an RGB monitor is required to display high-resolution graphics properly. In particular,

text in high-resolution mode, which is displayed 80 columns across the screen, is not usually legible on a television or color composite monitor. Because of this, in each of the sample programs, we first demonstrate the command, wait for a key to be pressed, then restore the screen to 40-column mode to ensure legibility.

HIGH RESOLUTION MODE

First, we must get into high-resolution mode. Start BASIC and type:

SCREEN 2 and press Enter

This selects high-resolution graphics mode. None of the other parameters to the **SCREEN** statement have any use when high resolution is selected with a mode of 2; multiple screens are not allowed, and because only black and white are available, the value for the burst parameter is irrelevant.

It follows that the **COLOR** statement is also irrelevant in high-resolution graphics mode. On the other hand, **CLS** and **KEY OFF** work as usual.

Black and white are the only colors available in high-resolution mode. White is the foreground (color 1), and black is the background (color 0). For compatibility with medium-resolution mode, color 2 is also black and color 3 is also white. In medium-resolution mode the default (foreground) color for the graphics statements is color 3, while in high-resolution mode the default color is color 1, white. Thus, **PSET (100,100)** plots a white dot at (100,100), and **PRESET (100,100)** plots a black dot, erasing the first.

The high-resolution graphics screen is 640 columns by 200 rows with the upper-left corner considered coordinate (0,0) and the lower-right corner (639,199). Both absolute and relative screen addressing modes can be used.

PSET

The **PSET** statement is the same in high-resolution mode as in medium-resolution mode except, of course, only the

colors black and white are available. For example, clear the screen in high-resolution mode and type:

PSET (320,100) and press Enter

to place a white dot at the center of the screen.

PSET in high-resolution mode differs from the medium-resolution mode version in one respect. In high-resolution mode, x values that are off the right edge of the screen (greater than 639) are not plotted; in medium-resolution mode, values off the right of the screen (greater than 319) wrap around to the left margin and appear on the screen.

The program in Listing 14-1 demonstrates the PSET statement in high-resolution graphics mode.

Listing 14-1. PSET Statement in High-Resolution Mode.

```
100 REM Program to demonstrate the PSET statement in
110 REM   high-resolution mode.
120 SCREEN 2:KEY OFF:CLS           'Set hi-res screen
130 REM Diagonal line down & right
140 FOR I=50 TO 150
150   PSET(I,I)
160 NEXT I
170 REM Diagonal line up & right
180 FOR I=50 TO 150
190   PSET(I*2+50,200-I)
200 NEXT I
210 LOCATE 24,27:PRINT "PRESS ANY KEY TO CONTINUE";
220 A$=INKEY$:IF A$="" THEN 220
230 SCREEN 0,1:WIDTH 40           'Reset screen
240 END
```

POINT

The POINT function in high-resolution mode returns either 0 (black) or 1 (white) for the color of the pixel checked.

The program in Listing 14-2 demonstrates the POINT function in high-resolution graphics mode.

LINE

The LINE statement in high-resolution mode is the same as in medium-resolution mode, with the exception of one

Listing 14-2. POINT Function in High-Resolution Mode.

```

100 REM Program to demonstrate the POINT function in
110 REM high-resolution mode.
120 SCREEN 2:KEY OFF:CLS 'Set hi-res screen
130 REM Draw vertical line
140 FOR I=50 TO 150
150 PSET(300,I)
160 NEXT I
170 REM Starting in column 50, draw a line to the right
180 REM until it hits the first line
190 COLUMN=50
200 COLUMN=COLUMN+1
210 IF POINT (COLUMN,100)<>0 THEN 240
220 PSET(COLUMN,100):PRESET(COLUMN-1,100)
230 GOTO 200
240 LOCATE 24,27:PRINT "PRESS ANY KEY TO CONTINUE";
250 A$=INKEY$:IF A$="" THEN 250
260 SCREEN 0,1:WIDTH 40 'Reset screen
270 END

```

error-handling characteristic. In medium-resolution mode, x values greater than 319, which are off the screen, wrap to the next line; in high-resolution mode, x values off the screen (greater than 639) are treated as 639.

Listing 14-3 demonstrates the LINE statement in high-resolution mode.

Listing 14-3. Line Statement in High-Resolution Mode.

```

100 REM Program to demonstrate the LINE statement in
110 REM high-resolution mode.
120 SCREEN 2:KEY OFF:CLS 'Set hi-res screen
130 REM Draw right-angle lines
140 LINE (10,50)-(10,100)
150 LINE (10,100)-STEP(629,0)
160 LINE (10,50)-(150,100)
170 REM Draw angled lines down to the base line
180 LINE (10,50)-(300,100)
190 LINE (10,50)-(450,100)
200 LINE (10,50)-(639,100)
210 REM Draw a box
220 LINE (10,110)-STEP(160,80),,B
230 REM Draw a solid box
240 LINE (340,130)-(460,170),,BF
250 LOCATE 24,27:PRINT "PRESS ANY KEY TO CONTINUE";
260 A$=INKEY$:IF A$="" THEN 260
270 SCREEN 0,1:WIDTH 40 'Reset screen
280 END

```


CIRCLE

Using the CIRCLE statement in high-resolution mode is only slightly different from using the statement in medium-resolution mode.

First, to produce the same images, all aspect ratios in high-resolution mode should be only half the value of those used in medium-resolution mode. The default aspect ratio in high-resolution mode, for example, is 5/12, rather than the 5/6 default of medium-resolution mode. In high-resolution mode, an aspect ratio of 5/12 produces a circle.

Second, it takes significantly longer to draw a circle of a given size in high-resolution mode than it does in medium-resolution mode. Because CIRCLE is a fairly slow-acting statement, this can be a major consideration.

Finally, although circles drawn in high-resolution mode can be colored black and white only, the edges are less jagged than their medium-resolution counterparts. This can be especially important because CIRCLE produces jagged edges, most noticeably with aspect ratios not equal to one.

When running the demonstration program in Listing 14-4, notice the smooth lines in the concentric ellipses.

Listing 14-4. CIRCLE Statement in High-Resolution Mode.

```
100 REM Program to demonstrate the CIRCLE statement
110 REM   in high-resolution graphics mode.
120 SCREEN 2:KEY OFF:CLS      'Set hi-res screen
130 REM Draw concentric ellipses
140 FOR RADIUS=10 TO 250 STEP 15
150   CIRCLE(320,100),RADIUS,1,,.3
160 NEXT RADIUS
170 LOCATE 24,28:PRINT "PRESS ANY KEY TO CONTINUE";
180 A$=INKEY$:IF A$="" THEN 180
190 SCREEN 0,1:WIDTH 40      'Reset screen
200 END
```

PAINT

The PAINT statement in high-resolution mode is **PAINT (x,y),paint** where **x** and **y** are the coordinates of the starting point and **paint** is equal to 0 or 1 to PAINT in black or white. There is no sense in specifying a boundary color, as

we do in medium-resolution mode; because there are only two colors, boundary must be the same as paint.

Like the CIRCLE statement, the PAINT statement is fairly slow and takes longer to execute in high-resolution mode than in medium-resolution mode.

The PAINT statement is used in the program shown in Listing 14-5, which demonstrates the GET and PUT statements in high-resolution mode.

GET And PUT

The GET and PUT statements are different in high-resolution mode only in the required size of the array that stores the image. In high-resolution mode, the formula that will roughly calculate the required size of the integer array that stores the image, while leaving a little extra space to be safe, is $\text{INT}((x/16)+1)*y+2$. (See Chapter 10 for definitions of x and y .) For a more exact formula, see the BASIC manual, but, unless you are constrained for space, the rough formula will serve perfectly well.

The program in Listing 14-5 demonstrates the GET and PUT statements in high-resolution mode.

DRAW

The DRAW statement works in the same fashion in high-resolution mode as in medium-resolution mode, except the horizontal distance covered by a given number of pixels is halved and only colors 0 and 1 are available.

Remember that the A subcommand to the DRAW statement corrects for aspect ratio in medium-resolution mode when a figure is rotated 90 or 270 degrees. The A subcommand also corrects for aspect ratio in high-resolution mode, but the correction factor is different. Because the aspect ratio of the high-resolution screen is 5/12, a ratio of 12/5 is used for correction when the figure is rotated. For example, clear the screen in high-resolution mode and type:

```
A$="U50 R120 D50 L120" and press Enter
```

to set the figure for what will appear to be a square in high-resolution mode, and then type:

```
DRAW "A0"+A$ and press Enter
```


Listing 14-5. GET and PUT Statements in High-Resolution Mode.

```

100 REM Program to demonstrate the GET and PUT graphics
110 REM statements in high-resolution graphics mode.
120 SCREEN 2:KEY OFF:CLS 'Set hi-res screen
130 DIM BALL(100)
140 REM Draw and save ball image
150 CIRCLE (2,2),2
160 PAINT STEP(0,0)
170 GET (0,0)-(4,4),BALL
180 CLS
190 REM Draw barriers
200 LINE (200,10)-(210,100),,BF
210 LINE (400,10)-(410,100),,BF
220 REM Set initial location & direction
230 BX=300:BDIR=-2
240 REM Put ball on screen at initial location
250 PUT (BX,50),BALL
260 REM Move the bouncing ball 300 times
270 FOR I=1 TO 300
280   BXOLD=BX
290   REM Trial move ball
300   BX=BX+BDIR
310   REM If hit barrier, reverse direction
320   IF POINT(BX+2,50)<>0 THEN BDIR=-BDIR:BX=BX+2*BDIR
330   REM Old ball off
340   PUT (BXOLD,50),BALL
350   REM New ball on
360   PUT (BX,50),BALL
370 NEXT I
380 LOCATE 24,27:PRINT "PRESS ANY KEY TO CONTINUE";
390 A$=INKEY$:IF A$="" THEN 390
400 SCREEN 0,1:WIDTH 40 'Reset screen
410 END

```

DRAW "A1"+A\$ and press Enter

DRAW "A2"+A\$ and press Enter

DRAW "A3"+A\$ and press Enter

to rotate the square 0, 90, 180, and 270 degrees. Note that all rotations are corrected so the figure still appears to be a square.

The program shown in Listing 14-6 demonstrates the DRAW statement in high-resolution mode.

HIGH RESOLUTION VERSUS MEDIUM RESOLUTION

On some other computers, high-resolution mode is *the* graphics mode, the mode in which all the best displays are

Listing 14-6. DRAW Statement in High-Resolution Mode.

```

100 REM Program to demonstrate the DRAW statement in
110 REM   high-resolution mode.
120 SCREEN 2:KEY OFF:CLS           'Set hi-res screen
130 REM Define rectangle
140 RECT$="u30 r15 d30 115"
150 REM Draw rectangle in all four rotations
160 FOR I=0 TO 3
170   COL=I*150+50                'Select column to draw box at
180   DRAW "bm=col;,100 a=i; x rect$;"
190 NEXT I
200 LOCATE 24,27:PRINT "PRESS ANY KEY TO CONTINUE";
210 AS=INKEY$:IF AS="" THEN 210
220 SCREEN 0,1:WIDTH 40           'Reset screen
230 END

```

produced. This is not true on the IBM PC. High-resolution mode has the most detailed picture, but, by and large, high-resolution mode is not particularly different from medium-resolution mode. If you need detail and have a good display, high-resolution mode is the best choice. High-resolution mode is also the only way to display 80-column text easily while in graphics mode.

On the other hand, medium-resolution mode displays better on televisions and composite monitors, is faster, and has the advantage of color.

The choice between the two modes is often dictated by the application. When it is not, the two modes are not so different that the choice is critical.

CHAPTER 15

A FUNCTION-GRAPHING PROGRAM

One of the best applications of high-resolution graphics mode is the plotting of detailed graphs. High-resolution mode allows a maximum number of points to be put on the screen so that smooth-edged graphs can be produced. In this chapter, we will develop a program that leads the user through the process of generating a high-resolution graph of a mathematical function.

A GRAPHING PACKAGE

Our graphing package plots a function on the high-resolution graphics screen. A *function* is a mathematical formula which for every value of x produces one and only one value of y . The range of x values is provided by the user. The x and y values, in conjunction with scales that cover the range of x and y , describe the points to be plotted on the graph. For example, $y=x*3$ is a function, where an x of 10 produces a y of 30, and so on. If we plotted a number of values of x and y , we would produce a graph like that shown in Fig. 15-1.

Our goal is to write a program that will plot a function. The user will select appropriate labels for the graph and the x and y axes and will specify a range of x values. The program will calculate the y values that correspond to the x values, will scale the y axis based on the calculations, and will plot each of the x,y pairs, producing a detailed

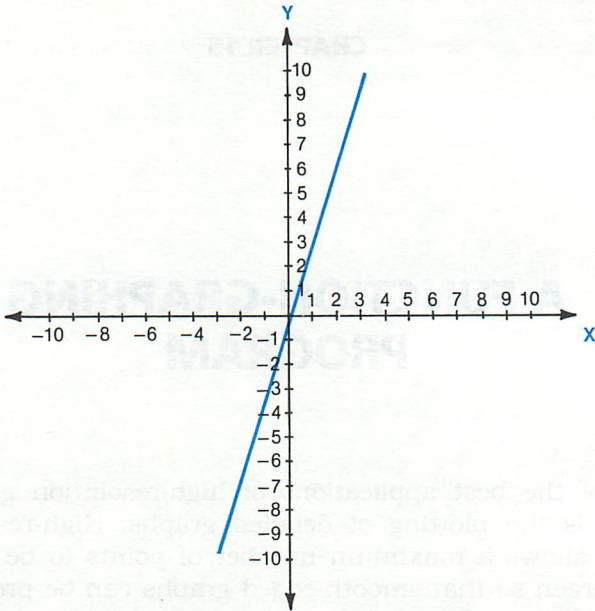


Fig. 15-1. Function $Y=X*3$.

graph of the function. The function will be integral to the program; that is, the user will only be able to change the function by changing a line in the program. The function used will be the sine function, a trigonometric function that produces a smoothly curved plot. The BASIC function which returns the sine of x is $\text{SIN}(x)$.

THE PROGRAM

The graphing program is shown in Listing 15-1. Enter and run the program now, so you can get a feel for how it operates. While the program may look a bit long to type, many of the lines are REM statements which you can omit. Good minimum and maximum values for x are 0 and 6.28, respectively. A good title is "SINE PLOT." The x and y axes can be titled "X" and "SINE(X)", respectively.

Note the smooth curve produced. Also note that the y axis is scaled to match the function; sine is always in the range -1 to 1 , and the y axis is scaled accordingly.

Listing 15-1. Plot Specified Function in High-Resolution Mode.

```

100 REM Program to plot a specified function in
110 REM   high-resolution mode.
120 REM Variables starting with I-L are integers
130 DEFINT I-L
140 REM Set high-resolution screen
150 SCREEN 2:KEY OFF:CLS
160 REM Prompt for labels
170 INPUT "Label for Graph? (max 78 letters) ",G$:
   G$=LEFT$(G$,78)
180 INPUT "Label for X axis? (max 45 letters) ",X$:
   X$=LEFT$(X$,45)
190 INPUT "Label for Y axis? (max 8 letters) ",Y$:
   Y$=LEFT$(Y$,8)
200 REM Prompt for range of x values
210 INPUT "Minimum X value (-10000 to 10000)? ",MINX
220 IF MINX<-10000 OR MINX>10000 THEN 210
230 INPUT "Maximum X value (-10000 to 10000)? ",MAXX
240 IF MAXX<-10000 OR MAXX>10000 THEN 230
250 REM Remove dialogue from screen
260 CLS
270 REM Print labels, centered
280 LOCATE 3,(80-LEN(G$))/2:PRINT G$
290 LOCATE 21,(50-LEN(X$))/2+21:PRINT X$
300 LOCATE 11,1:PRINT Y$
310 REM Put axes on screen. Y axis is double width
320 REM   to show well on poor-quality displays
330 LINE (154,36)-(155,136),,B:LINE -(554,136)
340 REM Draw hatch marks on axes
350 FOR I=0 TO 9
360   REM Hatch marks on y axis
370   LINE (152,36+10*I)-STEP(5,0)
380   REM Hatch marks on x axis, doubled to show up well
390   LINE (554-40*I,135)-STEP(1,2),,B
400 NEXT I
410 LOCATE 24,25:PRINT "PLEASE WAIT...I'M CALCULATING";
420 REM Calculate full set of y data points. All must be
430 REM   calculated before plotting so that program can
440 REM   automatically handle the range for the y scale
450 REM DATAPNT holds all y values
460 DIM DATAPNT(400)
470 REM Set min and max y so that any value will replace them
480 MINY=1.7E+38
490 MAXY=-1.7E+38
500 REM We will plot 401 points-figure x increment
510 REM   for one point
520 XINC=(MAXX-MINX)/400!
530 REM Calculate all y values, store, and determine
540 REM   y max and min
550 FOR I=0 TO 400

```

Listing 15-1—cont. Plot Specified Function In High-Resolution Mode.

```

560 REM Determine x value for data point # I, based on XINC,
570 REM then use subroutine at line 470 to calc y value
580 XVALUE=MINX+I*XINC:GOSUB 900:DATAPNT(I)=YVALUE
590 REM Check for max & min
600 IF YVALUE>MAXY THEN MAXY=YVALUE
610 IF YVALUE<MINY THEN MINY=YVALUE
620 NEXT I
630 REM Print scales on corresponding axes. MINX and MAXX
640 REM were input by the user
650 LOCATE 5,5:PRINT "Max ";:PRINT USING "#####.###";MAXY
660 LOCATE 17,5:PRINT "Min ";:PRINT USING "#####.###";MINY
670 LOCATE 19,15:PRINT "Min ";:PRINT USING "#####.###";MINX
680 LOCATE 19,65:PRINT "Max ";:PRINT USING "#####.###";MAXX
690 REM YINC is number of y units each pixel represents
700 YINC=(MAXY-MINY)/1001
710 REM Loop to plot points
720 FOR I=0 TO 400
730 REM Figure how many pixels up from x axis to plot
740 REM Special case-if MAXY=MINY, then always plot
750 REM 50 pixels up
760 IF YINC<>0 THEN HEIGHTINPIXELS=(DATAPNT(I)-MINY)/YINC
ELSE HEIGHTINPIXELS=50
770 REM Plot point. First time, simply plot dot, thereafter,
780 REM draw a line from last point to current point
790 IF I<>0 THEN LINE -(155+I,136-HEIGHTINPIXELS) ELSE
PSET(155+I,136-HEIGHTINPIXELS)
800 NEXT I
810 REM Clear any keystrokes, so user doesn't end by accident
820 IF INKEY$<>"" THEN 820
830 REM Wait for a keystroke before ending, so can admire
840 LOCATE 24,25:PRINT " PRESS ANY KEY TO CONTINUE ";
850 A$=INKEY$:IF A$="" GOTO 850
860 SCREEN 0,1:WIDTH 40 'Reset the screen
870 END
880 REM Subroutine to calculate YVALUE given XVALUE
890 REM Change line 870 to change function
900 YVALUE=SIN(XVALUE)
910 RETURN

```

We will now cover the program a line at a time. One general point to be made is that very little of the program produces graphics; most of the lines handle text input and output or perform mathematical calculations in support of the graphics. This is usually true in graphics work. Plotting a point is easy—knowing *where* to plot the point is the hard part!

Setting Up

Line 130 defines all variables starting with the letters I through L as integers. This means that I will be an integer, while MAXX will be a real number. We will use integers where possible because BASIC processes them faster than real numbers, but, because some of our calculations involve fractional numbers, we cannot use integers for everything.

Line 150 initializes the high-resolution graphics screen by selecting high-resolution mode, turning off the soft keys, and clearing the screen.

User Input

We need several pieces of information from the user. Lines 170-190 prompt the user for the labels for the graph, x axis, and y axis. These labels will appear on the final screen, so the display can be identified and interpreted readily.

Lines 210 and 230 prompt the user for the range of x values to be plotted. The graph is always the same size in pixels, 101 rows by 401 columns. The computer converts the user-specified range of x-values so it can draw the graph using 401 horizontal pixels. In later lines, the program will automatically compensate for the selected x values.

Line 260 clears the prompts and user input from the screen, so we can begin plotting the graph.

Setting up the Axes

The first step in drawing the graph is to lay out and label the axes. We will always place the axes and labels in the same places, except that the labels are moved as needed to center them.

Line 280 prints the label for the graph, centered between the left and right margins of the screen. The `LEN(G$)` function returns the length of the string `G$`, which holds the title, so that $(80 - \text{LEN}(G\$))/2$ calculates the column in which the label should start so as to be centered. (Remember that in high-resolution mode there are 80 columns of text across the screen.) Line 290 prints the

label for the x axis, similarly centered between columns 21 and 71. This is offset so the label will appear centered with respect to the x axis. Line 300 prints the label for the y axis. This label is not centered, but is started as far left as possible to leave the maximum amount of space available for text between the start of the label and the y axis.

Line 330 puts the x and y axes on the screen. The first LINE statement draws the y axis from row 36 down to row 136, and in columns 154 and 155, by drawing a box with corners (154,36) and (155,136). The doubled line is necessary so vertical lines will show as brightly as horizontal lines, particularly on poor-quality displays. The second LINE statement on line 330 draws the x axis from the lower end of the y axis to (554,136), so the x axis is 400 pixels long plus the point where it intersects the y axis. The x axis is drawn as a single, rather than double, line because it is horizontal.

The FOR . . . NEXT loop from lines 350 to 400 draws crosshatch marks on the x and y axes, dividing each of these axes into 10 equal parts. Line 370 draws 10 crosshatches 6 pixels long across the y axis, starting at row 36 (the top of the y axis), then row 46, 56, and so on down to 126. Line 390 draws 10 crosshatches 3 pixels long across the x axis, starting at column 554 (the end of the x axis), then column 514, and so on down to column 194. As with the y axis and all other vertical lines drawn, the vertical crosshatches are double width so they will show up well on all types of displays.

Evaluating the Function

We must calculate each value for y to determine the range of y values in order to scale the y axis. Because the required 401 calculations (one for each column on the graph) take a considerable period of time, we will also save the calculated values while determining the range. In this way we can perform two functions at once: scaling the y axis and calculating the set of y values.

We will store the 401 y values in the array DATAPNT, which has 401 storage locations set aside on line 460.

To determine the minimum and maximum y values, we first set the variables MINY and MAXY to very high and very low values, respectively, on lines 480 and 490. Then, any lower or higher value (for MINY and MAXY, respectively) we calculate will become the new minimum or maximum. The initial values are selected so the first value will replace them. This means that after processing one value that value is both the minimum and the maximum. Succeeding values will alter MINY and MAXY so the final values for these variables will indeed be the minimum and maximum values for y .

There will be 401 x values plotted, one for each column on our graph. We will divide evenly the range of x into 401 points, then use each of these 401 points to calculate the corresponding y value. Line 520 calculates how many x units there are between each horizontal pixel and stores this value in XINC. For example, if the range of x were from 100 to 900, or 800 units, then XINC would be 2. We have the initial column at the y axis, which would have the x value MINX, or 100. Then the next column's x value would be $100 + \text{XINC} = 102$, the next would be $100 + \text{XINC} * 2 = 104$, and the last (or 401st) column's x value would be $100 + 400 * \text{XINC} = 900$, which is equal to MAXX, as it should be. In general, the formula for the x value for data point number n is $x = \text{MINX} + \text{XINC} * n$. Each pixel across the graph represents XINC units, and all of them taken together cover the range from MINX to MAXX.

The FOR . . . NEXT loop at lines 550 to 620 uses XINC to calculate each of the 401 x values from MINX to MAXX, and then uses the x values to calculate the corresponding y values. Line 580 first figures the x value corresponding to column number I (temporarily stored in XVALUE), calls the subroutine on line 900 which calculates the y value (YVALUE) based on XVALUE, and then stores the y value in the DATAPNT array. Lines 600 and 610 check the current YVALUE to see if it is a new maximum or minimum value for y . At the end of this FOR . . . NEXT loop, all 401 y values and the y maximum and minimum values are determined. The information is ready to be graphed.

The Scales

The ends of the x and y axes must be labeled for numeric value, so the user can tell what values the different points plotted represent. The MINY and MAXY values we just calculated represent the range of y values and in lines 650 and 660 are printed at their respective ends of the y axis. The MINX and MAXX values previously input by the user are printed, on lines 670 and 680, at their respective ends of the x axis.

The PRINT USING statement is used so that exactly three fractional digits are printed. The pound signs in the format string indicate that three *and only three* digits to the right of the decimal point should be printed, and that places for up to six digits to the left of the decimal point should be reserved. Letting BASIC print as many digits as needed with a simple PRINT statement could result in as few as one digit or as many as 8 digits being printed, spoiling the appearance of the graph and possibly printing into the working area. For example, type:

```
PRINT 1: PRINT 10/3 and press Enter
```

and you will see why it is necessary to use the PRINT USING statement to force a fixed number of decimal places.

Plotting the Points

On line 700, YINC, the y axis counterpart to XINC, is calculated. YINC is the number of y units represented by each pixel along the y axis.

We are now prepared to step through each of the 401 x values and plot the corresponding point. This is done in the FOR . . . NEXT loop from lines 720 to 800.

The height, in pixels, of each point above the x axis is calculated in line 760, and the value stored in the variable HEIGHTINPIXELS. This is given by the formula $\text{HEIGHTINPIXELS} = (\text{DATAPNT}(I) - \text{MINY}) / \text{YINC}$, which returns the portion of the total vertical range of 101 pixels represented by the given data point. There is one special case, however. If $\text{MINY} = \text{MAXY}$, that is, if the function is a horizontal line, then YINC will be zero, so the above equa-

tion would result in division by zero and thus produce a BASIC error message. To handle this special case, we arbitrarily put every point 50 pixels above the x axis. These two cases are handled by the IF . . . THEN . . . ELSE statement in line 760.

Line 790 plots the data point. The column is 0 through 400 columns away from the left end of the x axis, as specified by the formula **column=155+I** where **I** is the number of the current data point. That is, the first data point is in column 155 (the left end of the x axis), the second is in column 156, and the 401st point is in column $155 + 400 = 555$. The row is **HEIGHTINPIXELS** pixels above the x axis, as given by the formula **row = 136 - HEIGHTINPIXELS** where **HEIGHTINPIXELS** is the number of pixels that the current data point is raised above the x axis.

The first point is PSET on the screen. Following points are plotted with the LINE statement, with the first coordinate omitted so that a line is drawn from the preceding point to the current point. This produces a continuous plot. The IF . . . THEN . . . ELSE statement in line 790 determines whether the point should be plotted with PSET or LINE.

The graph is now complete!

Finishing

Lines 820-870 end the program. Line 820 clears any pending keystrokes. This is important because the program takes a long time to run, and the user might press a key either accidentally or out of impatience. Line 850 then waits for a keystroke before ending, so the user can take as long as he wants to examine the graph. Line 860 restores the screen to 40-column text mode. This is done only because high-resolution mode produces 80-column text which is illegible on some displays. Line 870 ends the program.

The Function

Lines 900 and 910 comprise the subroutine which calculates the y value given the x value and the function being

plotted. The statement `YVALUE=SIN(XVALUE)` on line 900 does all the actual calculation of the sine of x . We might replace line 900 with `YVALUE=SQR(XVALUE)` to plot the square root of x , or `YVALUE=XVALUE ^ 3` to plot the third power of x . Try these or any other function, and see for yourself that any function can be plotted by inserting the proper calculation of `YVALUE` as a function of `XVALUE`.

Line 910 ends the subroutine and directs the program to resume at the statement immediately following the `GOSUB 900` statement that called the subroutine.

SUMMARY AND ENHANCEMENTS

The graphing program just discussed is mostly complete. The user is fully prompted for input, and a smooth, labeled plot is produced. Because the calculations are lengthy, it might be a good idea to notify the user of progress through the calculation section so he does not think the program has gone into an infinite loop. This could be done with a display in the corner that says "Processing number n " where n is the number of the current data point.

A useful addition would be to allow the user simply to type the function, rather than having to alter the program every time the function is to be changed. This would, however, take quite a bit of programming.

The lesson to be learned here is primarily that high-resolution mode can produce extremely detailed graphics. The steps we've followed to scale and plot the graph apply to the design of any graphing program. Such programs are most useful when they are able to handle varying ranges, functions, and output sizes, as well as input from and output to disk files. A method of plotting and differentiating multiple functions on the same set of axes would also be highly useful. In medium-resolution mode, the various functions could be set apart with color, while in high-resolution mode, the addition of dotted or dashed lines would accomplish the purpose.

TEXT-MODE GRAPHICS

Some good graphics effects can be produced with the text characters available on the IBM PC. Text mode is not, strictly speaking, a "graphics mode"; however, the text characters available on the PC include not only the normal keyboard letters and numbers, but also shapes and symbols which can be combined in a mosaic fashion to produce pleasing displays.

The resolution available in text mode is a little hard to define. Each of the characters drawn by the Color/Graphics Adapter is formed from an 8-by-8 matrix of pixels. When using a screen 40 characters in width, the resolution of the screen in terms of characters is only 25 rows by 40 columns, but the resolution in terms of the pixels comprising the characters is (25×8) by (40×8) , which is 200 by 320, exactly the same as in medium-resolution mode. Similarly, when the screen is 80 columns wide, the resolution in terms of pixels is 200 by 640, just as in high-resolution mode. Therefore, in those applications where the available characters can be put together to make the smooth lines you require, you have the same resolution at your disposal as in the graphics modes.

One distinct advantage of text mode is that all 16 colors of the IBM PC are available at all times. Text mode is the only mode in which this is true. (Even when the screen contains 80 characters across, and the resolution is effectively the same as high-resolution mode, all 16 colors are available.) There are limitations, however. The character itself can be any one of the 16 colors, and the background

for the rectangle which contains the character can be any one of the 8 low-intensity colors, but each character can be composed of only these two colors. This means that only two colors can appear within each 8-by-8 character box. Text mode, therefore, requires ingenuity both to get pixels of several colors close together and to avoid the appearance of blockiness.

Another advantage of text mode is that it is fast, because 64 pixels (the 8-by-8 character box) can be drawn simply by putting a single character on the screen. Text in text mode appears much faster than does text in the graphics modes, where the computer must draw each character one pixel at a time. Also, in many ways, text mode is more convenient than the graphics modes, because only two statements, LOCATE and PRINT, are required to produce graphics rather than the somewhat intimidating array of graphics commands available in the graphics modes.

Text mode has another advantage over the graphics modes in the ability to have what are effectively 4 to 8 separate screens, any one of which can be displayed and any one of which can be worked with at any time. This means that the process of altering or redrawing the screen can be hidden from the viewer, so changes can be presented in a movie-frame fashion with one complete screen projected after the other. This advantage would be more important if there wasn't an often overriding need for compatibility with the monochrome display.

The monochrome display (which is run off the Monochrome Adapter and is completely separate from the Color/Graphics Adapter) has absolutely no graphics capability *per se*. It does, however, display the same character set as does the Color/Graphics Adapter. Although the characters on the monochrome display are formed from more and smaller pixels, the result of combining them to produce graphics-like effects is much the same. Graphics done in text mode for the Color/Graphics Adapter will work well on the monochrome display; consequently, all IBM PC users can run a program that uses only text-mode graphics. (The graphics will, however, be without color on the monochrome display.)

Unfortunately, the multiple screens available in text mode on the Color/Graphics Adapter use extra memory because it is required for the graphics modes. Because there are no graphics modes on the monochrome screen, there is no extra memory on the Monochrome Adapter, and there are no multiple screens available when using the monochrome display. The result is that virtually no one uses the multiple-page potential of the Color/Graphics Adapter in programs they plan to sell or distribute, because they are unwilling to lose compatibility with the large monochrome display market. Of course, you can use multiple screens for your own programs, but it is not advisable to do this in programs you plan to sell or share with others who might not have a Color/Graphics Adapter.

Another consideration is that the monochrome screen has no 40-column mode; programs written for 40-column text mode on the Color/Graphics Adapter will display on the left half of the monochrome screen. Because 80-column text mode does not display well on television sets, there can be a problem in programs written for the mass market.

Text-mode graphics applications are not that different from the uses of the graphics modes. Text mode does bar charts well, but does not do continuous graphs as well. Pie charts are not a good application, because there is no way to form pie-wedge shapes from the available character set. Arcade-style games can and have been done in text mode; certainly text-mode games can be colorful and flicker-free.

Why didn't we cover text-mode graphics first in this tutorial? And why bother with the complication of the graphics modes? Well, text-mode graphics are essentially a patchwork affair—capable of producing good results, but only in limited situations and with considerable sleight of hand. The more complex the application, the more apparent this will become. For example, there is no text-mode equivalent of the PUT command. For small objects, this is no problem, but producing large objects in text mode requires multiple LOCATE and PRINT statements. Image locations are limited because characters can only appear every 8 pixels, and because character locations can be

specified only with absolute, rather than relative, coordinates.

Furthermore, the smallest distance a moving object can travel is one character position, producing jerky animation. Finally, many desirable shapes, such as circles, cannot be drawn using the characters provided on the PC.

Text-mode graphics are neither as flexible nor as easy to use in complicated situations as are graphics-mode graphics. On the other hand, they have advantages where speed, 16 colors, multiple pages, and monochrome compatibility are important. The choice between the two depends on the application, but remember that the features which set both the IBM PC and Advanced BASIC apart in terms of graphics are primarily available only in the graphics modes and are the reason for the existence of the Color/Graphics Adapter.

SETTING UP THE TEXT-MODE SCREEN

Three statements are required to configure the screen for text mode—**SCREEN**, **WIDTH**, and **COLOR**. Two of these statements are familiar from our description of the graphics modes but assume a new form in text mode.

The command **SCREEN 0,1** selects color-text mode, and **SCREEN 0,0** selects black-and-white text mode. As mentioned previously, the **SCREEN** statement is of the general form **SCREEN mode,burst,activepage,visualpage** where **mode** selects the screen mode, **burst** enables or disables color, and **activepage** and **visualpage** select which among the multiple screens are to be displayed and worked with. **Visualpage** defaults to **activepage** if not otherwise specified.

A mode of 0 selects text mode. A burst of 0 selects black-and-white mode, while a nonzero burst enables color. (This operation of burst in text mode is the reverse of that in medium-resolution graphics mode.) As with medium-resolution graphics mode, burst does not disable color on RGB monitors; in fact, it does not appear to have any effect whatsoever on RGB monitors in text mode. All parameters are optional, and, if omitted, are left at the pre-

vious values. Note that if either mode or burst is changed, the screen is cleared, the foreground color is set to white, and the background and screen border are set to black.

The `activepage` and `visualpage` parameters are both fascinating and strange. For example, get into text mode on the Color/Graphics Adapter by typing:

```
SCREEN 0,0 and press Enter
```

and then type:

```
SCREEN,,1,1 and press Enter
```

All previous text will vanish; you are on a new screen called page 1. (Of course, the display is on the same screen, but the selectable page that the screen “shows” has been changed.) Type:

```
SCREEN,,0,0 and press Enter
```

and your first screen, page 0, will reappear unchanged, but missing the characters that were typed while you were on page 1. The switch is instantaneous.

So far, all seems clear, but now type:

```
SCREEN,,1,0 and press Enter
```

and then type:

```
PRINT "THIS IS PAGE 1" and press Enter
```

Don't worry—we haven't destroyed either your keyboard or your screen. The reason nothing is appearing on the screen is that we have set the active page—the screen to which all output goes—to page 1. However, the visual page has not been changed from page 0, and so you do not see your input as it is “echoed” to the screen. To see the input, press Esc to clear the line you are on (because you can't see it, there's no telling what might be there) and then type:

```
SCREEN,,,1 and press Enter
```

even though you can't see what you've typed. There is the message “THIS IS PAGE 1” and the rest of the lines you've typed. It can be confusing to use multiple screens, but it can be extremely convenient as well. The viewer

need never see a display while it is being constructed and can be presented with several screens in rapid succession—all the screens can be prepared first, then shown one right after the other with no speed constraint. Of course, there is potential for a major problem if the program user is accidentally left on the wrong screen, or if input is typed when the active page and visual page differ. Care is required in the use of active and visual pages.

The values for `activepage` and `visualpage` can vary from 0 to 7 when the screen width is 40 columns (more on this when we discuss the `WIDTH` statement), and from 0 to 3 when the screen width is 80 columns. Any attempt to select an invalid page while using the Color/Graphics Adapter will result in an "Illegal function call" error. There is one curiosity here, however. When on the monochrome display, error conditions may not occur as expected. For example, in a system with both displays, no error occurs when the display is on the monochrome display and pages other than 0 are selected. If page 3 is selected, and if the graphics display has been initialized by being used during that session, the text will appear on the graphics screen even though you're using the monochrome screen. We suggest that you make sure that you are not on the monochrome screen before using pages other than page 0. We will discuss, in Chapter 19, means by which your programs can check for the current display screen.

There is one other important point concerning the use of multiple pages. You might reasonably assume that there is a cursor associated with each screen, so that when you return to that screen, the cursor will be where you left it. You would, unfortunately, be incorrect. The cursor position is carried from one screen to the next. For example, select text mode, clear the screen with `CLS`, and type:

```
SCREEN,,1,1 and press Enter
```

```
SCREEN,,2,2 and press Enter
```

```
SCREEN,,3,3 and press Enter
```

and finally:

```
SCREEN,,0,0 and press Enter
```


to return to the original screen. Notice how the cursor progresses down the screen as it moves through the various pages. There is a way to save the cursor location when leaving a screen and then to restore it upon return. The function **z=CSRLIN** returns the row number of the cursor, while the function **z=POS(y)** returns the column the cursor is in. The parameter **x** to the **POS** function serves no purpose whatsoever; any variable or constant of any value may serve as this parameter. Thus the command **OLDROW=CSRLIN: OLDCOLUMN=POS(1)** can save the current cursor position in the variables **OLDROW** and **OLDCOLUMN** prior to leaving a given screen. Upon returning to this screen, the cursor can be restored with **LOCATE OLDROW,OLDCOLUMN**. (We will cover the **LOCATE** statement in more detail later.)

In general, **POS(x)** and **CSRLIN** are useful for telling where the cursor is on the screen. An equivalent function in the graphics modes, to report the last point referenced, is lacking. **CSRLIN** is not actually a function; rather, it is a special BASIC variable, like **INKEY\$**, that has a value set by BASIC rather than by the program. **POS(x)**, on the other hand, is a true function. There is no practical difference between functions and special variables except that special variables have no parameters and thus cannot be passed values. In the case of **POS(x)**, the **x** parameter is a "dummy," and so **POS(x)** might just as well have been a special variable.

The WIDTH Statement

Setting the number of columns displayed on the screen is accomplished with the **WIDTH** statement. Type:

```
WIDTH 40:PRINT "TEST 40" and press Enter
WIDTH 80:PRINT "TEST 80" and press Enter
```

and:

```
WIDTH 40:PRINT "TEST 40" and press Enter
```

Only 40 or 80 columns are allowed. Note that there are other forms of the **WIDTH** statement that have nothing to

do with the screen display, but rather affect files, printers, and asynchronous communication.

Incidentally, the **WIDTH** statement can be used in the graphics modes as well. However, in medium-resolution mode, text can only be displayed in 40 columns, and in high-resolution mode, text can only be shown in 80 columns. Consequently, the **WIDTH** command must force the screen into the graphics mode appropriate to the selected width if the current mode is not appropriate. If the current mode is medium resolution and **WIDTH 40** is typed, nothing happens, but if **WIDTH 80** is entered, the screen is cleared and set to high-resolution mode. Similarly, if the current mode is high resolution and **WIDTH 80** is entered, nothing happens, but if **WIDTH 40** is entered, the screen is cleared and set to medium-resolution mode with color turned on and palette 1 selected. Thus **WIDTH** can serve as a shorthand way to switch between certain graphics modes.

As we mentioned at the beginning of this chapter, the monochrome screen can display only 80-column text. No error will be reported if **WIDTH 40** is executed on the monochrome screen; after this command, text will display only on the left half of the monochrome screen (columns 1-40). Lines longer than 40 columns will wrap around to the next line. Any attempt to use the **LOCATE** statement to set the cursor past column 40 will result in an "Illegal function call" error. One oddity: the monochrome screen cannot be set to a width of 40 until a **SCREEN 0** statement is executed. The width simply will not change until this is done.

We will primarily use 40-column text mode in our examples, because this mode works well on all displays.

COLOR in Text Mode

There are three color regions in text mode: foreground, background, and border. The *foreground* is the body of the character displayed. The *background* is the part of the 8-by-8 pixel box the character does not occupy, as well as the whole 8-by-8 box for those locations that are blank. The *border* is the area around the working area of the

screen. In graphics modes, the border is always set to the background color, but in text mode, we can control this area separately and can produce attractively framed displays, particularly on RGB monitors.

The color of each region is controlled with the **COLOR** statement. In text mode the **COLOR** statement is **COLOR foreground,background,border** where **foreground**, **background**, and **border** refer to the body of the character, the area the character is printed against, and the edge of the screen, respectively, as discussed previously.

On the Color/Graphics Adapter, foreground may have any value in the range 0-31. Values of 0-15 indicate one of the 16 standard PC colors, as shown in Table 16-1, while adding 16 to any value produces the same foreground color but blinking. The valid values for background are 0-7, corresponding to the first 8 colors in Table 16-1. The value for border may be 0-15, representing the full 16 colors available, again as shown in Table 16-1. (It is possible to obtain the full 16-color set of background colors, but, because this is not provided by BASIC, we will defer discussion of it until Chapter 19.) Out-of-range values are often converted to correct values and used with no error message—for example, a background value of 8 is treated as 0.

Table 16-1. Colors Available on the IBM PC

NUMBER	COLOR*	NUMBER	COLOR*
0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	High-Intensity White

* Any of the 16 colors can be used for the foreground and/or border color; only colors 0-7 can be used for the background color.

Note that when the border value is changed, the color of the entire border immediately changes. However, when

the foreground and background values are changed, the effects only appear when new operations to the screen take place. That is, if you switch the background color from black to red, don't expect the color of all the background pixels to immediately switch to red; rather, when you PRINT a character, the background area within that character's 8-by-8 box will be red. Then, if you clear the screen, it will become red, because every character on the screen is redrawn as a blank.

For example, select text mode, clear the screen, and type:

```
COLOR,,4 and press Enter
```

The border immediately becomes entirely red. Now type:

```
COLOR,2 and press Enter
```

to select a green background and notice that only the background of the Ok prompt and any new characters you type are green. Type:

```
CLS and press Enter
```

and the screen becomes all green. Likewise, type:

```
COLOR 3,0 and press Enter
```

to select cyan text on a black background. Only those characters typed after the COLOR statement are cyan on black.

There is a good reason why all foreground and background colors remain the same until explicitly changed. If the COLOR statement changed the colors of every character on the screen, it would be possible to have only two colors on the screen at once—one foreground color and one background color. As it is, it is easy to get multiple colors on the screen. For example, type:

```
COLOR 1,0: PRINT "A": COLOR 2: PRINT "B":  
COLOR 4: PRINT "C" and press Enter
```

Background color can be changed as easily as foreground, but in this example, we leave the background colored black for the sake of legibility on television sets.

The program shown in Listing 16-1 demonstrates the multiple color combinations available in text mode in both 40- and 80-column widths. This program displays every text size and color combination possible on the PC. Type and run it, paying particular attention to the quality of your display. On a good RGB monitor, every character should be clear, while on an average television, most characters will not be legible in 80-column mode.

Listing 16-1. Color Text Modes.

```

100 REM A program that demonstrates the color text modes.
110 REM 80 character text mode may be illegible on TV's
120 REM and low-quality monitors. This program is useful
130 REM for testing the quality of a color monitor.
140 SCREEN 0,1:COLOR ,0:KEY OFF:CLS 'Set COLOR TEXT MODE
150 DIM A$(40) 'Set aside memory
160 A$="40 CHARACTER COLOR TEXT MODE"
170 WIDTH 40 'Set to 40 characters on line
180 LOCATE ,,0 'Turn cursor off
190 X=5 'Set printing location
200 GOSUB 290 'Display text on screen
210 A$="80 CHARACTER COLOR TEXT MODE"
220 COLOR ,0:WIDTH 80 '80 columns/black background
230 LOCATE ,,0 'Turn cursor off
240 X=25 'Set printing location
250 GOSUB 290 'Display text on screen
260 WIDTH 40 'Set back to 40 columns
270 COLOR 7,0,0:KEY ON:CLS 'Restore screen
280 END 'Return control to BASIC
290 FOR BG=0 TO 7 'Each line is printed in
300 LOCATE 5+BG,X ' a different color
310 FOR FG=1 TO LEN(A$) 'Characters will be 16
320 COLOR FG MOD 32,BG,4 ' different characters
330 PRINT MID$(A$,FG,1); 'Display the next character
340 NEXT FG 'Display characters until
350 PRINT ' done displaying all
360 NEXT BG ' eight lines
370 COLOR 7,0:LOCATE 18,X:PRINT "PRESS ANY KEY TO CONTINUE"
380 A$=INKEY$:IF A$="" THEN 380 'Wait for key to continue
390 RETURN 'Finished with this string

```

If your graphics display is a black-and-white screen, the colors will appear as various shades of gray, often with every other column white and the alternating columns dark.

There is a question concerning color in text mode that

may already have occurred to you. What happens on the monochrome screen? In general, the results are common-sense, but there are important exceptions to this rule.

If the foreground color of a character is 0-7, the character is normal intensity and white on black. Foreground colors 8-15 produce high-intensity white on black, colors 16-23 produce blinking white on black, and colors 24-31 produce high-intensity, blinking white on black.

The exceptions are as follows. Colors 1, 9, 17, and 25 cause the character to be underlined. For example, foreground color 25 produces a high-intensity, blinking, underlined white on black character.

Also, if the foreground color is any of 0, 8, 16, or 24 (varieties of black), and the background color is 0 (black), there will be no display (black on black). While this sounds pointless, it can be useful for applications such as typing hidden passwords. Oddly enough, if the foreground color is 7 (white), and the background color is also 7, the display is not white on white.

If the foreground color is 0, 8, 16, or 24 (black), and the background color is 7 (white), the display is black on white. Blinking and high-intensity still apply as usual; for example, **COLOR 16,7** sets the screen to display blinking black characters in a white background.

Consult the discussion of the **COLOR** statement in the IBM *BASIC* manual for a summary of color effects on the monochrome display.

USING TEXT MODE

Now that we have described the fundamentals of text mode, we can proceed to use text mode to draw graphics. The key commands in this connection are **LOCATE** and **PRINT**.

We discussed **LOCATE** in the chapter on text in medium-resolution graphics mode, but we only covered that portion of the statement we needed at the time. Besides moving the cursor, **LOCATE** can also alter the cursor itself. Clear the screen completely (remember that this is done with **CLS** and **KEY OFF**) and type:

LOCATE 10,10,0: FOR I=1 TO 2000: NEXT and press Enter

The cursor disappears for the duration of the **FOR . . . NEXT** loop and then reappears. (BASIC always produces a visible cursor when the program ends to let you know where the next characters you type will appear.) Type:

LOCATE 20,20,1,0,7 and press Enter

and the cursor will appear at its full height.

The full form of the **LOCATE** statement is **LOCATE row,column,cursor,start,stop**. **Row** is the screen line the cursor is to be put on (1 through 25), and **column** is the screen column (1 through 40 or 1 through 80, depending on the selected width). Note that the upper-left corner of the screen is (1,1), not (0,0) as in the graphics modes. A zero value for **cursor** will turn the cursor off, while a one will turn it on. Note that the cursor cannot be put on line 25 unless a **KEY OFF** statement has been executed.

Start and **stop** control the top and bottom edges of the cursor. In text mode, all characters are 8 lines high, as measured in pixels, and so the cursor is no more than a maximum of 8 lines high. The top line of the 8 is called scan line 0, and the bottom line is called scan line 7. Thus, the crossbar in a capital T would be toward scan line 0, and the descender in a lowercase y would be toward scan line 7. **Start** is the scan line at which the cursor should start appearing white, and **stop** is the scan line at which it should cease to appear white. Type:

LOCATE,,,0,0 and press Enter

to make the cursor a sort of "overline" character, and:

LOCATE,,,3,4 and press Enter

to move it to the center of the character it is over. We can even type **LOCATE,,,5,2** to wrap the cursor around from the bottom to the top of the line Type:

LOCATE,,,6,7 and press Enter

to restore the cursor to its original state.

All the parameters to the **LOCATE** statement are

optional. The value of any omitted parameter will remain unchanged.

The PRINT statement is straightforward in operation. The syntax of PRINT is **PRINT expressions**; where **expressions** can be any numeric, string, or logical variables or constants, and the optional trailing semicolon leaves the cursor at the end of the last expression printed, preventing the cursor from moving to the next row as it will otherwise. (A comma may also be specified to move the cursor to the next print field, but is of no relevance here.) We will PRINT only strings, because we are interested in characters that produce graphics effects, rather than numeric values. We will usually use the trailing semicolon, because, if BASIC is on lines 24 or 25 and tries to go to the next line, the entire display will scroll up, moving the top line off the screen and displacing all graphics one line.

Graphics effects can be achieved using only typewriter keyboard characters. In the days when most computers had only hard copy output, the many variants on the popular *Star Trek* game used the asterisk, parentheses, plus sign, uppercase I, and underscore to draw an entire universe (or at least a galaxy). Many good posters were done with typewriter characters and overstrikes.

We can produce graphics of this kind on the PC. For example, clear the screen and type:

```
FOR I=1 TO 22: LOCATE I,I: PRINT "++++++  
++++++": NEXT I and press Enter
```

to draw a slanted grid. Type:

```
CLS: FOR I=1 TO 38: LOCATE 10,I: PRINT " o";:  
NEXT I and press Enter
```

to make a ball appear to move across the screen.

The program in Listing 16-2 demonstrates one of the ways in which the normal characters may be used for graphics (see Fig. 13 in the color photograph section). While the graph as drawn is colorful and perfectly adequate for many applications, it is rather crude. The typewriter character set is simply inadequate for most purposes.

Listing 16-2. Text-Mode Graphics.

```

100 REM Program to demonstrate text-mode graphics with
110 REM   a simple graphing program.
120 SCREEN 0,1:COLOR 1,0,0:WIDTH 40:KEY OFF:CLS
130 LOCATE ,,0           'Cursor off
140 REM Draw axes
150 FOR I=1 TO 20:LOCATE I,6:PRINT "I":NEXT I
160 LOCATE 20,7,0:PRINT STRING$(33,"_")
170 REM Label axes
180 COLOR 2
190 LOCATE 8,3:PRINT "S":LOCATE 9,3:PRINT "I"
200 LOCATE 10,3:PRINT "N":LOCATE 11,3:PRINT "E"
210 LOCATE 13,2:PRINT "(x)"
220 LOCATE 22,22:PRINT "x (Radians)"
230 REM Plot a number of sine points
240 COLOR 4
250 FOR X=0 TO 12 STEP .4
260   COLUMN=X*2.5+7      'Column to plot in
270   SINEX=SIN(X)        'Calc sine value
280   REM Calc row that sine value plots in
290   ROW=15-(SINEX+1)*5
300   LOCATE ROW,COLUMN:PRINT "*";
310 NEXT X
320 COLOR 7:LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
330 A$=INKEY$:IF A$="" THEN 330
340 COLOR 7,0,0:CLS      'Reset screen
350 END

```

Once again, the PC has what we need: a set of unusual but useful characters, such as smiling faces, musical notes, card suits, and arrows, as well as a set of characters designed especially for text-mode graphics. The latter set consists of multiple forms of single and double lines which can be connected to form a seamless network, solid blocks of several shapes, and three shadings that are 25%, 50%, and 75% solid. The complete character set available from BASIC is shown in Appendix A.

You will notice that most of the characters are designed for special purposes such as mathematical notation and foreign language applications, and have no particular use for graphics. The most useful graphics characters fall into two distinct sets, numbers 1-32 and 176-223.

Because they are not present on the keyboard, these characters cannot be typed between quotes. How do we print them? The function `z=CHR$(x)` converts the

numeric value in *x* into its string equivalent. The numeric values of the characters are listed next to the characters in Appendix A. For example, `PRINT CHR$(1)` prints character number 1, the smiling face, and `PRINT CHR$(236)` prints character number 236, the infinity sign.

The characters 1-32 are special characters with useful shapes. The arrows (numbers 23-27) make ideal pointers or missiles, the sun (number 15) makes a good star, the smiling faces (numbers 1 and 2) make balls, targets, and symbols of all kinds, and the space (number 32) is the all-important blank. There are a number of characters that do not print but have some other effect instead. For example, character number 7 causes the speaker to beep but does not print. These special-function characters can be useful in certain cases; in particular, characters 28-31 move the cursor without using the `LOCATE` statement, and character 11 homes the cursor to the upper-left corner. To see an example, clear the screen and type:

```
PRINT CHR$(11): FOR I=1 TO 10:  
PRINT CHR$(28);CHR$(31);"";: NEXT and press Enter
```

The first command homes the cursor, and each of the ten times through the `FOR . . . NEXT` loop moves the cursor right, then down, then prints a slanted line. The end result is a dashed line slanted toward the lower-right corner.

Incidentally, the special-function characters displace some of the special characters available on the PC. There are 11 characters that exist but cannot easily be printed from BASIC, including the universal male and female symbols and another musical note. Printing these requires use of the ROM in the PC; this will be described further in Chapter 19.

The other set of special characters, numbers 176-223, produce the cleanest text-mode graphics for large areas. For example, the program shown in Listing 16-3 draws two concentric boxes as smoothly as they could be drawn in the graphics modes. Note, however, that these boxes are drawn as close together as is possible in text mode. The program shown in Listing 16-4 produces color shading (see Fig. 12 in the color photograph section), and the

program shown in Listing 16-5 uses the block and half-block characters 219-223 to construct a simple maze. Each of these is as good as any graphics-mode drawing for special purposes. The limitations are that characters can only be located every 8 pixels, and that only the limited pixel arrangements provided by the characters are available. For example, there is no way to produce curved or jagged lines easily in text mode, or to make the spacing between lines less than 8 pixels, as, for example, in Listing 16-3.

Listing 16-3. Concentric Boxes in Text Mode.

```

100 REM Program to draw concentric boxes in text mode.
110 SCREEN 0,1:COLOR 7,0,0:WIDTH 40:LOCATE ,,0:KEY OFF:CLS
120 REM Draw edges of outer box
130 FOR I=5 TO 35:LOCATE 5,I:PRINT CHR$(196):NEXT I
140 FOR I=5 TO 35:LOCATE 20,I:PRINT CHR$(196):NEXT I
150 FOR I=6 TO 19:LOCATE I,4:PRINT CHR$(179):NEXT I
160 FOR I=6 TO 19:LOCATE I,36:PRINT CHR$(179):NEXT I
170 REM Draw corners of outer box
180 LOCATE 5,4:PRINT CHR$(218)
190 LOCATE 5,36:PRINT CHR$(191)
200 LOCATE 20,36:PRINT CHR$(217)
210 LOCATE 20,4:PRINT CHR$(192)
220 REM Draw edges of inner box
230 FOR I=6 TO 34:LOCATE 6,I:PRINT CHR$(196):NEXT I
240 FOR I=6 TO 34:LOCATE 19,I:PRINT CHR$(196):NEXT I
250 FOR I=7 TO 18:LOCATE I,5:PRINT CHR$(179):NEXT I
260 FOR I=7 TO 18:LOCATE I,35:PRINT CHR$(179):NEXT I
270 REM Draw corners of inner box
280 LOCATE 6,5:PRINT CHR$(218)
290 LOCATE 6,35:PRINT CHR$(191)
300 LOCATE 19,35:PRINT CHR$(217)
310 LOCATE 19,5:PRINT CHR$(192)
320 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
330 A$=INKEY$:IF A$="" THEN 330 ELSE CLS
340 END

```

Type and run the program in Listing 16-6. This program prints each of the 255 characters available from BASIC. (This character set is also shown in Appendix A.) Some of the characters are blank, one beeps, and one clears the screen; these are special function characters.

The semicolon we have been using at the end of the PRINT statement deserves further attention. If the semico-

Listing 16-4. Text-Mode Color Shading.

```

100 REM Program to demonstrate text mode color shading.
110 SCREEN 0,1:COLOR ,0,0:WIDTH 40:LOCATE , ,0:KEY OFF:CLS
120 REM Select each of colors 1-7 in turn
130 FOR I=1 TO 7
140   COLOR I
150   REM Select each of 25%, 50%, & 75% shading characters
160   FOR J=176 TO 178
170     REM Draw a strip across the screen in each shading
180     REM   and color
190     FOR COLUMN=5 TO 35
200       LOCATE ,COLUMN:PRINT CHR$(J);
210     NEXT COLUMN
220     PRINT
230   NEXT J
240 NEXT I
250 LOCATE 24,8:PRINT "PRESS ANY KEY TO CONTINUE";
260 A$=INKEY$:IF A$="" THEN 260 ELSE CLS
270 END

```

Listing 16-5. Block and Half-Block Characters in Text Mode.

```

100 REM Program to demonstrate block & half block
110 REM   characters in text mode.
120 SCREEN 0,1:COLOR 2,0,0:WIDTH 40:LOCATE , ,0:KEY OFF:CLS
130 REM Draw edges
140 FOR I=10 TO 30:LOCATE 2,I:PRINT CHR$(220):NEXT
150 FOR I=4 TO 21:LOCATE 1,10:PRINT CHR$(221):NEXT
160 FOR I=10 TO 28:LOCATE 22,I:PRINT CHR$(223):NEXT
170 FOR I=3 TO 21:LOCATE 1,30:PRINT CHR$(222):NEXT
180 REM Draw long barriers
190 FOR I=3 TO 14:LOCATE 1,15:PRINT CHR$(222):NEXT
200 FOR I=10 TO 21:LOCATE 1,22:PRINT CHR$(221):NEXT
210 FOR I=15 TO 17:LOCATE 1,19:PRINT CHR$(222):NEXT
220 FOR I=7 TO 9:LOCATE 1,18:PRINT CHR$(221):NEXT
230 REM Cross barriers
240 FOR I=16 TO 19:LOCATE 14,I:PRINT CHR$(220):NEXT
250 FOR I=18 TO 21:LOCATE 10,I:PRINT CHR$(223):NEXT
260 FOR I=13 TO 19:LOCATE 18,I:PRINT CHR$(223):NEXT
270 FOR I=18 TO 27:LOCATE 6,I:PRINT CHR$(220):NEXT
280 FOR I=10 TO 17:LOCATE 16,I:PRINT CHR$(219):NEXT
290 FOR I=20 TO 30:LOCATE 8,I:PRINT CHR$(219):NEXT
300 FOR I=17 TO 20:LOCATE 12,I:PRINT CHR$(219):NEXT
310 COLOR 7 'Back to white letters
320 LOCATE 25,8:PRINT "PRESS ANY KEY TO CONTINUE";
330 A$=INKEY$:IF A$="" THEN 330 ELSE CLS
340 END

```


Listing 16-6. Text Character Set.

```

100 REM Program to demonstrate complete text character set.
110 REM Initialize text mode screen
120 SCREEN 0,1:COLOR 7,0,0:WIDTH 40:LOCATE ,,0:KEY OFF:CLS
130 REM Run through ASCII values 1-255 (full char set)
140 FOR I=1 TO 255
150   REM Print char with selected ASCII value
160   PRINT CHR$(I);" ";:IF POS(X)>=38 THEN PRINT:PRINT
170   FOR J=1 TO 100:NEXT J           'Short delay
180 NEXT I
190 PRINT:LOCATE 25,8:PRINT "PRESS ANY KEY TO CONTINUE";
200 A$=INKEY$:IF A$="" THEN 200 ELSE CLS
210 END

```

lon is not used, the cursor proceeds to the left end of the next row after printing. This is not necessarily a problem; however, if the cursor is on row 24 or row 25 when the PRINT without a semicolon occurs, then the entire screen scrolls up. This means that every row on the screen is moved up one row, with the top row lost. For example, type:

```
LOCATE 24,1: PRINT "PRINTING ON ROW 24" and press
Enter
```

and note that the printed text is immediately moved to row 23 as the screen scrolls. The text on row 25 does not normally scroll, even when a KEY OFF statement has been executed. However, executing a PRINT without a trailing semicolon on row 25 does cause rows 1 through 24 to scroll.

For most graphics applications, scrolling is best avoided by appending a semicolon to all PRINT statements. Although, if you have a requirement for animation that involves moving the entire screen up, scrolling can be handy. The next chapter demonstrates such an application.

THE SCREEN FUNCTION

There is a text-mode counterpart to the POINT graphics function. The SCREEN function (as distinguished from the SCREEN statement) returns either the character at a loca-

tion or the *attribute* of that character, where the attribute consists of the foreground and background colors, as well as blinking characteristics.

The SCREEN function is **x = SCREEN (row, column, attributeflag)** where **row** and **column** are the location to be tested, specified as for the LOCATE statement, and **attributeflag** indicates whether the attribute or character is to be returned. If attributeflag is omitted or 0, the ASCII value of the character (as shown in Appendices A and B) is returned, while a nonzero attributeflag causes the attribute of the character to be returned.

Using the SCREEN function to determine which character is present on the screen is straightforward. Type:

```
LOCATE 10,10: PRINT "A";: PRINT
SCREEN(10,10) and press Enter
```

This PRINTs a capital A at (10,10), then checks for its value. If you look at Appendix A, you will see that the value returned, 65, is indeed the ASCII value of capital A. Another way to demonstrate this is with the CHR\$ function. Remember that this function converts an ASCII value into its character equivalent. Hence, if we enter **LOCATE 10,10: PRINT "A";: PRINT CHR\$(SCREEN(10,10))**, we get another capital A. Here we read the character's ASCII value, 65, out of the screen, and then convert it back into a character and print it.

Interpreting the value returned for the attribute is a little more complicated. There are three values returned in one number: foreground color, background color, and blink. These are organized within a single byte as shown in Fig. 16-1. If this figure is unclear to you, there is a simple procedure for interpreting attribute values.

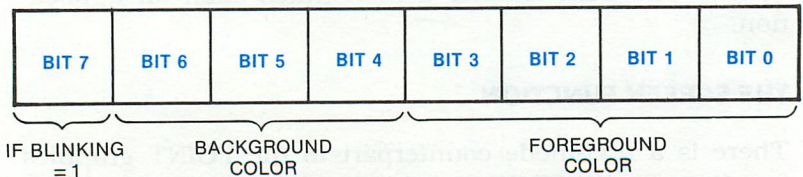


Fig. 16-1. Text-mode attribute byte organization.

First, the attribute must be obtained. Clear the screen and type:

```
COLOR 7,2
```

to select white lettering on a green background. (If this is illegible on your screen, you may select other colors, but, in that case, your results from here on will differ from ours.) Type:

```
LOCATE 10,10: PRINT "A";:
A=SCREEN(10,10,1) and press Enter
```

to put a capital A on the screen at (10,10), then store the value of the letter's attribute in the variable A.

Now type:

```
COLOR 7,0: CLS: PRINT "FOREGROUND: ";(A AND 15)
and press Enter
PRINT "BACKGROUND: ";(A AND 112)/16 and press
Enter
IF (A AND 128) THEN PRINT "BLINKING" ELSE
PRINT "NOT BLINKING" and press Enter
```

The values of 7 for foreground, 2 for background, and nonblinking that we set with the COLOR statement will be shown. (The screen is cleared only for legibility.) We will describe these attributes one at a time.

The foreground value is calculated by the formula (A AND 15) where A is the attribute value. The value returned is in the range 0-15, corresponding to the 16 available colors.

The background color is calculated by (A AND 112)/16 where A contains the attribute value. The value returned is in the range 0-7, corresponding to the 8 low-intensity background colors.

If the character is blinking, (A AND 128) will return a nonzero value, which is to say that it will be interpreted as TRUE by an IF statement. If the character is not blinking, (A AND 128) will return zero, which is interpreted as FALSE. As shown above, an IF . . . THEN . . . ELSE statement can readily be used to check the blink.

The program shown in Listing 16-7 uses the SCREEN function to check the screen contents. This is our familiar

bouncing ball example, but with an important difference. The wall is composed entirely of solid block characters (ASCII value 219). Each wall is composed of blocks of a different color however. By using the SCREEN function to get the attribute byte of the block, then ANDing the attribute with 15 to get the foreground color, we can tell which wall we have hit. Notice that the program does not merely negate the direction when it hits; rather, it sets the direction based on the wall struck. This is only possible because the SCREEN function tells us precisely which wall has been hit. For the x direction, this is done on lines 310-330, and for the y direction it is done on lines 380-400.

The SCREEN function performs in precisely the same fashion on the monochrome screen as it does with the Color/Graphics Adapter. The foreground, background, and blink values selected via the COLOR statement are returned in the attribute value and can be separated as previously described. Of course, these values must be interpreted according to the rules of the monochrome screen. For example, a color value of 1 for foreground indicates "blue" on the Color/Graphics Adapter, but indicates "underlined" on the monochrome screen.

MULTIPLE SCREEN PAGES

We can now properly demonstrate the active and visual screen pages available in text mode. The program shown in Listing 16-8 first draws four different faces in four different active screen pages. These four screen pages are then alternated rapidly; the successive images create the impression of motion. Animation of this sort is rapid and virtually flickerfree. This listing is quite long; if you do not type it, at least look over the program to get a sense of the use of multiple screen pages.

TEXT-MODE GRAPHICS

We would like to leave you with a sense of both the advantages and disadvantages of text-mode graphics. Text-

Listing 16-7. SCREEN Function.

```

100 REM Program to demonstrate use of the
110 REM SCREEN function.
120 REM Set text mode screen
130 SCREEN 0,1:COLOR 7,0,0:WIDTH 40:KEY OFF:CLS:LOCATE ,,0
140 REM Draw walls, each in a different color
150 COLOR 1:FOR I=10 TO 30:LOCATE 2,I:PRINT CHR$(219):NEXT
160 COLOR 2:FOR I=10 TO 30:LOCATE 22,I:PRINT CHR$(219):NEXT
170 COLOR 3:FOR I=3 TO 21:LOCATE 1,10:PRINT CHR$(219):NEXT
180 COLOR 4:FOR I=3 TO 21:LOCATE 1,30:PRINT CHR$(219):NEXT
190 REM Initialize the ball
200 COLOR 7,0,0
210 BX=20:BY=5
220 BXINC=1:BYINC=1
230 LOCATE BY,BX:PRINT "O"
240 REM Loop to move ball
250 FOR I=1 TO 200
260 REM save old location
270 BXOLD=BX:BYOLD=BY
280 REM Move in x direction
290 BX=BX+BXINC
300 REM Check for collision & reverse if needed
310 IF SCREEN(BY,BX) = 32 THEN 360 'If space, then didn't hit
320 IF (SCREEN(BY,BX,1) AND 15) = 3 THEN BXINC=1 'Hit left
330 IF (SCREEN(BY,BX,1) AND 15) = 4 THEN BXINC=-1 'Hit right
340 BX=BX+2*BXINC 'Undo old move & make new one
350 REM Move in y direction
360 BY=BY+BYINC
370 REM Check for collision & reverse if needed
380 IF SCREEN(BY,BX) = 32 THEN 420 'If space, then didn't hit
390 IF (SCREEN(BY,BX,1) AND 15) = 1 THEN BYINC=1 'Hit top
400 IF (SCREEN(BY,BX,1) AND 15) = 2 THEN BYINC=-1 'Hit bottom
410 BY=BY+2*BYINC 'Undo old move & make new one
420 LOCATE BYOLD,BXOLD:PRINT " "; 'Blank old ball off
430 LOCATE BY,BX:PRINT "O"; 'New ball on
440 NEXT I
450 LOCATE 24,8:PRINT "PRESS ANY KEY TO CONTINUE";
460 AS=INKEY$:IF AS="" THEN 460 ELSE CLS
470 END

```

mode graphics are colorful, fast, and easy to use for certain purposes. Animation is, however, jerky; this is inescapable when the shortest possible movement is 8 pixels. Text-mode displays are also typically angular and somewhat blocky. On the other hand, the screen is more colorful than is possible in the graphics modes, and text-mode programs are relatively straightforward and easy to understand or debug—and, of course, will run on any IBM PC display.

Listing 16-8. Multiple Pages in Text-Mode Animation.

```

100 REM Program to demonstrate the use of multiple pages
110 REM   in text mode animation.
120 SCREEN 0,1,0,0:COLOR 7,0,0:WIDTH 40:KEY OFF:CLS
130 LOCATE ,,0 'Turn cursor off
140 REM Screen 0 face
150 SCREEN ,,0,0:CLS:GOSUB 480 'Draw basic face
160 COLOR 5:LOCATE 6,15:PRINT "O" 'Left eye
170 LOCATE 6,25:PRINT "O" 'Right eye
180 LOCATE 13,20:PRINT CHR$(127) 'Nose
190 REM Screen 1 face
200 SCREEN ,,1,1:CLS:GOSUB 480 'Draw basic face
210 COLOR 5:LOCATE 8,13:PRINT "O" 'Left eye
220 LOCATE 8,23:PRINT "O" 'Right eye
230 LOCATE 13,19:PRINT CHR$(127) 'Nose
240 REM Screen 2 face
250 SCREEN ,,2,2:CLS:GOSUB 480 'Draw basic face
260 COLOR 5:LOCATE 10,15:PRINT "O" 'Left eye
270 LOCATE 10,25:PRINT "O" 'Right eye
280 LOCATE 13,20:PRINT CHR$(127) 'Nose
290 REM Screen 3 face
300 SCREEN ,,3,3:CLS:GOSUB 480 'Draw basic face
310 COLOR 5:LOCATE 8,17:PRINT "O" 'Left eye
320 LOCATE 8,27:PRINT "O" 'Right eye
330 LOCATE 13,21:PRINT CHR$(127) 'Nose
340 REM Animate by alternating faces
350 REM Animate 30 times
360 FOR TIMES=1 TO 10
370   REM Run through all four pages
380   FOR I=0 TO 3
390     SCREEN ,,I,I 'Select new screen
400     FOR J=1 TO 100:NEXT J 'Delay a bit
410   NEXT I
420 NEXT TIMES
430 SCREEN ,,0,0 'Back to page 0
440 COLOR 7,0,0 'Reset color
450 LOCATE 24,8:PRINT "PRESS ANY KEY TO CONTINUE";
460 A$=INKEY$:IF A$="" THEN 460 ELSE CLS
470 END
480 REM Subroutine to draw face
490 REM Draw outline
500 COLOR 1 'Blue outline
510 FOR I=11 TO 29:LOCATE 2,I:PRINT CHR$(196):NEXT I
520 FOR I=11 TO 29:LOCATE 22,I:PRINT CHR$(196):NEXT I
530 FOR I=3 TO 21:LOCATE I,10:PRINT CHR$(179):NEXT I
540 FOR I=3 TO 21:LOCATE I,30:PRINT CHR$(179):NEXT I
550 LOCATE 2,10:PRINT CHR$(218)
560 LOCATE 2,30:PRINT CHR$(191)
570 LOCATE 22,30:PRINT CHR$(217)
580 LOCATE 22,10:PRINT CHR$(192)
590 REM Hair
600 COLOR 14 'Yellow hair

```


Listing 16-8—cont. Multiple Pages in Text-Mode Animation.

```

610 LOCATE 1,19:PRINT "\|/"
620 REM Mouth
630 COLOR 12                                'Red mouth
640 FOR I=16 TO 24:LOCATE 16,I:PRINT CHR$(205):NEXT
650 LOCATE 16,15:PRINT CHR$(212)
660 LOCATE 16,25:PRINT CHR$(190)
670 COLOR 9                                'Blue eyes
680 REM Right eye
690 FOR I=13 TO 17:LOCATE 5,I:PRINT CHR$(196):NEXT
700 FOR I=13 TO 17:LOCATE 11,I:PRINT CHR$(196):NEXT
710 FOR I=6 TO 10:LOCATE 1,12:PRINT CHR$(179):NEXT
720 FOR I=6 TO 10:LOCATE 1,18:PRINT CHR$(179):NEXT
730 LOCATE 5,12:PRINT CHR$(218)
740 LOCATE 11,12:PRINT CHR$(192)
750 LOCATE 11,18:PRINT CHR$(217)
760 LOCATE 5,18:PRINT CHR$(191)
770 REM Left eye
780 FOR I=23 TO 27:LOCATE 5,I:PRINT CHR$(196):NEXT
790 FOR I=23 TO 27:LOCATE 11,I:PRINT CHR$(196):NEXT
800 FOR I=6 TO 10:LOCATE 1,22:PRINT CHR$(179):NEXT
810 FOR I=6 TO 10:LOCATE 1,28:PRINT CHR$(179):NEXT
820 LOCATE 5,22:PRINT CHR$(218)
830 LOCATE 11,22:PRINT CHR$(192)
840 LOCATE 11,28:PRINT CHR$(217)
850 LOCATE 5,28:PRINT CHR$(191)
860 RETURN

```

The choice between modes is yours, and, as the next chapter will show, text-mode graphics can be powerful indeed. Nonetheless, we suspect that as you do more graphics on the PC, you will develop a preference for the graphics modes.

RACECAR—AN ARCADE-STYLE GAME

Now we will put what we've learned about text mode graphics to use. We will design a game called *Racecar*. We will follow the game-design approach we used in Chapter 11 for *Blockbuster*, but there are differences between the designs of the two. This is, in part, a function of the differences between text and graphics modes; however, the primary reason is that each game is a work in and of itself, the design of which cannot be reduced to a set of step-by-step instructions.

RACECAR

The synopsis of *Racecar* follows. The player tries to guide a car along a constantly weaving roadway, staying to the middle of the road and avoiding the sides. If either side of the road is hit, a nonfatal crash occurs. The roadway has a fixed length, and the object of *Racecar* is to drive the full length of the road with as few crashes as possible.

All graphics will be done in text mode.

There are three sections to design—the road, the car, and “housekeeping.” The road divides into two further sections, the initial road and the moving road.

As we design and program *Racecar*, it will become apparent that we had a good idea of what the game consisted of before we began to design. The game-making process is not so neatly defined as indicated by the follow-

ing discussion, and it is impossible to design a game unless you already have an overall concept. As much of the game as possible should be worked out before you begin programming.

The Initial Road

Before the action part of the game begins, we must set up the playing field. This involves positioning a stretch of road on the screen before play begins. This stretch of road orients the player and avoids the appearance of a road that begins from nowhere. The initial stretch of road is always straight and centered, so the player will not be confused at the start of the game and so the car can be started in the center every time.

Before discussing the road in detail, we will establish the basics of the display. We have already specified that the graphics consist of text-mode characters. In order to avoid undue complexity, we do not use the type of text-mode graphics that involve combining special characters together to create images; rather, we use only normal typewriter-style characters, so the basic unit size is one character.

The screen width is 40 characters. While 80 characters across the screen would allow for more detailed graphics, 80-character mode does not display well on television sets. So the game will be playable on all displays, we use 40 characters per row. Note that if the game is played on the monochrome screen, it will display in the left 40 columns only.

We use black and white text mode and only black (color 0) and white (color 7). These colors provide legibility and clarity on poor-quality displays.

We are now ready to specify the dimensions of the roadway. A space of ten columns (one-fourth of the screen) seems a good first choice. However, an odd value for the number of columns is preferable, so the car can be set squarely in the middle of the roadway. In actual use (including some testing and correction), nine columns proved effective.

The first choice to be made in designing the roadway is

to select a character to delineate the sides. Thinking ahead, however, we decide that the easiest way to tell if the car has hit the shoulder is to use the SCREEN function. If the contents of the screen where the car is about to move are not blank, then the shoulder has been hit.

But while we know that a shoulder has been hit, we don't know which one. Why do we care which one? Well, thinking still further ahead, because we have decided that a crash is not fatal, we need to "restart" the car after each crash. One option is to restart it in the center of the roadway. However, we prefer to have the car "bounce" off the side back into the road. In this case, it is critical to know which shoulder is hit, because the car must bounce back in the direction of the roadway, safely clear of the shoulder, before restarting.

Given this, it is necessary to use a different character for each side. Then the value returned by the SCREEN function can be used, not only to tell whether a shoulder has been hit, but also which shoulder. Thus far, we have determined that the road is outlined with two sides, each drawn with a different character. A problem remains: what if as the road is angling to the left, the player is moving the car to the right. Because there is a combined motion of two character positions before checking for a collision, it is possible for the car to be "driven" between the characters of the shoulder without hitting one. The player would be off the road, the program would be out of control, and the player would be able to drive the car off the screen.

The solution is to use the width of two characters, rather than one, to draw the side of the road. Also, the car and the road cannot be allowed to shift by more than one character width between collision checks, so that driving outside the road is impossible.

We now know the road is surrounded with two shoulders composed of different characters. The shoulders are each two characters in width. We have also discovered more about *Racecar* in the process of designing the road. The parts of this game cannot be designed in isolation; in fully outlining one section, we must consider other sections as well.

This interdependence of parts is often true in game design. At best, you will have a strong concept of the game throughout the design phase and will be able to keep the many parts straight in your head. More likely, however, you will do your best at design, start to program the game, think of possible problems, go back to the design phase to make changes, and program again. While you are testing the program, you will find other problems. (For example, we did not catch the car driving unharmed through the single-width shoulder until we tested the program.) Remember, too, that we are only talking about implementing the design concept; conceiving the design concept is sometimes more difficult.

To summarize, you must make the pieces fit as best you can. As you try to sort out the ramifications of one section, you may find that your task grows out of control; take notes, follow an outline, and think.

We are now ready to complete the design of the initial road. We know that the initial road is composed of two sides in a straight line, extending the length of the screen. The left shoulder is composed of the right-bracket character (]), and the right shoulder is made of the left-bracket character ([), because these characters look suitably like a boundary wall.

The Moving Road

The play of the game takes place entirely against the backdrop of a moving road. To make the road appear to be more than one screen long, we keep the car in the center of the screen at all times, and make the road appear to move past it.

The direction of the road's motion is a problem. Usually, one would expect that the road would move toward the bottom of the screen to disappear below (and in effect behind) the car. However, there is only one way that the screen scrolls with any speed in BASIC, and that is up. Because we have decided that the car remains fixed, and the road moves, and because the road may wander anywhere on the screen, we must program the road to appear to move up and off the screen, with the car effectively

moving downward. As discussed in Chapter 16, we can make the screen scroll by executing a PRINT statement, without a semicolon, on row 24 or row 25 of the screen.

We add to the bottom of the road with the same PRINT statement with which we scroll the screen. The effect will be to move the top section of the road off the screen, the new road section onto the bottom of the screen, and the road sections in between up one row.

We know how wide the road is and of what characters it is composed. We need to know where to place each new road section. We can only let the road wander one column to the right or left from its previous position, because the player couldn't handle sharper turns, and if the road moved two or more columns, the shoulder would have gaps through which the car could slip. Remember that each road section will first appear on row 24. Because the car is in the center of the screen, it does not encounter any given road section until that section has been scrolled up several times.

The road shoulder will be moved only one column from its previous position each time a new section is added to the bottom of the road. The direction of motion is randomly selected, so the road is generated "spontaneously" during play of the game.

If we let the road wander randomly as each row is drawn, it will move too abruptly for the player to respond; worse, it will tend to "jitter" rather than weave through broad curves. To make the road weave well, we execute each direction of wander several times before randomly selecting a new one. The number of times to wander in each selected direction (the *wander factor*) before choosing a new one is best determined by trial and error. We arrived at three repetitions as an appropriate value for the wander factor.

In moving the road left and right across the screen, we must check for the left and right margins of the screen, so the road doesn't wander off the edge. We do this by performing a trial move in the selected direction; if the new column is at either margin, we reverse the direction and perform a double move in the new direction. This double

move undoes the trial move and makes one normal move away from the margin.

The Car

The car is represented by a single character on the screen. The character representing the car will stay on the same row, row 10, as the road appears to move by it. When the screen scrolls to move the road up, the character representing the car will scroll up as well. We could erase the scrolled car character before redrawing the car on row 10, but this would result in jerky motion. Also, by leaving the character on the screen, we can create a trail of the car's motion on the screen to help give the player a sense of his current motion. For these reasons, the car's track is left on the screen to trail towards and off the top.

The car, like the road, can wander only one column each time the screen is scrolled one row. A greater motion would make the car difficult to control. Keyboard input will be used to select the direction of the car's motion—right, straight, or left. We could require that a key be pressed each time the car is to be moved one column, but continued motion in the specified direction until another key is pressed usually provides better control.

Motion of the car into the shoulders of the road must be monitored. Because a previous section of the program ensures that the road cannot wander off the screen, the car can only hit the shoulder or not hit the shoulder; there are no other possibilities. If the car does not hit the shoulder, it is drawn in its new location.

If the car hits the shoulder, we first determine which shoulder by checking the character on the screen that the car has hit. We then pause, make a crash sound by executing the BEEP statement three times, and pause the action long enough to let the player prepare to resume the game.

There are several options for restarting the car. One choice is to restart it in the same way it started when the game began. Another is to put the car in the middle of the road. We prefer to make the car appear to “bounce” off the shoulder by moving it away from the shoulder and restarting it. We move the car two columns away from the shoul-

der so it doesn't hit the shoulder again on the next move. This is why it is important to know which shoulder we've hit; we need to know which way to move to avoid the shoulder.

We do not draw the car until we have checked for collision with the shoulder, so we can't accidentally erase part of the shoulder.

The car starts in the middle of the road, headed straight ahead. This properly belongs in "housekeeping," but is necessary to start the car so it can be tested.

We will use the up-arrow character (^), above the number 6 at the top of the keyboard) to represent the car.

Housekeeping

One task of housekeeping is deciding how long the race should be. We use 200 repetitions, which, in conjunction with the repeat factor of three for drawing road sections in each randomly selected direction, draws 600 road sections in each game.

We must keep track of the number of crashes. This number is initialized and displayed on the screen before the game is run. Each time the car crashes, the number of crashes is updated on the screen. When the game ends, nothing more need be done because the number of crashes is already displayed.

PROGRAMMING RACECAR

Racecar is now completely outlined. We will program each section of the game separately, making sure each part works before proceeding to the next. Because we described modular programming in detail in Chapter 11, we will not go into more detail here. Also, as described in Chapter 11, there are no comments in the program listings, and there are multiple statements on many lines because of speed considerations. A listing with comments is shown at the end of this chapter.

Racecar is programmed in four steps, corresponding to the four sections described above. A listing is provided with each section. Note that the earlier listings seem to

have gaps in the line numbering. The missing numbers are reserved for lines dealing with later steps. You do not need to leave such gaps in your own programs because you can use the RENUM statement to make line numbers orderly. We space the line numbers as we do for explanatory purposes.

Programming the Initial Road

The program in Listing 17-1 sets up the screen and the initial road. Line 100 sets up the screen for *Racecar*. Black and white text mode is selected, with a white foreground against a black background and border. The screen is then cleared. The DEFINT statement sets all variables to default to integers, because operations on integers are performed much more quickly than those on floating-point numbers, and we need only integers.

Line 120 defines the width of the track (stored in variable TRKWIDTH) as nine columns between the shoulders. The initial position of the left shoulder of the road, stored in TRKPOS, is set to column 15, which centers the road on the screen.

The initial road is put on the screen in line 150. Twenty-three road sections are drawn; the FOR . . . NEXT loop does each section in turn. The sections are drawn by using LOCATE to position the cursor at column TRKPOS (the left shoulder column), and row 24. Two right brackets (the left shoulder), the number of spaces indicated by TRKWIDTH, and two left brackets (the right shoulder) are then PRINTed. (The SPC function produces the number of spaces indicated by the parameter, in this case 9, as specified by TRKWIDTH.)

This PRINT statement creates one road section. The PRINT does not have a trailing semicolon, and because it is on row 24, the entire screen scrolls up one line, clearing room for the next road section. After this has been done 23 times, the road extends from the top of the screen to the bottom in a straight line.

One note: row 25 does not scroll when we print on row 24. This will become important when we do house-keeping.

Listing 17-1. Racecar—Initial Screen.

```
10 REM Racecar—initial screen.
100 DEFINT A-Z:SCREEN 0,0:COLOR 7,0,0:WIDTH 40:LOCATE ,,0:
    CLS:KEY OFF
120 TRKWIDTH=9:TRKPOS=15
150 FOR I=1 TO 23:LOCATE 24,TRKPOS:
    PRINT "]" SPC(TRKWIDTH) "[" :NEXT I
320 END
```

Programming the Moving Road

The program shown in Listing 17-2 sets up the screen and then moves the road.

As described in the design section, the wandering of the road from side to side is randomly selected. There is a BASIC function, RND, which is designed to return a sequence of random numbers, one number each time it is called. However, RND must be initialized with the RANDOMIZE statement, which is of the form **RANDOMIZE(n)** where **n** is a number used to seed the random string returned by RND. Unfortunately, the same string is returned every time the same value for **n** is used for a seed. We want some way to get a different seed every game so each game is different from the rest.

One approach would be to use the statement **RANDOMIZE** with no **n**. In this case, the player is asked for a seed. However, we would rather have the seeding process invisible to the player. We can do this with the special BASIC variable **TIME\$**.

The value of **TIME\$** is always the current time in HH:MM:SS format. If the time was set at boot-up, then **TIME\$** returns the correct time; if not, it returns the time elapsed since the computer was booted. In either case, the seconds portion of **TIME\$** will serve well as a random seed with values between 0 and 59. The seconds are characters in a string, but we can easily convert them to a numeric value to use in the **RANDOMIZE** statement.

First, the **RIGHT\$** function, which selects only the speci-

fied rightmost portion of a string, can be used to separate the seconds portion from the TIME\$ string. Type:

```
A$=TIME$: PRINT A$: PRINT RIGHT$(A$,2) and press
Enter
```

to see how this works. Then the VAL function, which converts numbers in string form to numeric values, can be used to turn the seconds string into a number suitable to seed RANDOMIZE. Understand that a number such as 3 is what we need; what the RIGHT\$ function by itself returns is a string like "03" which cannot be used for numeric operations. For example, 3+2 is the number 5, but "2"+"03" is the string "203".

Line 110 initializes the RND function based on the time. Because there are only 60 possible values for the seconds string, there are only 60 possible racecourses. This is plenty to create the impression of great variety, as it seems unlikely any player would recognize a repeat course among 60 different ones. If necessary, however, we could also pick out the value of the minutes portion of the TIME\$ string, multiply it by 60, and add it to the value of the seconds portion to give a much greater range of possible seed values and, thus, courses.

Line 130 sets NUMLINES, the wander factor, to 3. (The wander factor is the number of times that each randomly selected direction of movement for the track is repeated before a new direction is selected.) The track movement process is repeated 200 times; the variable TRKLEN is the counter for this. Therefore, 600 road sections are drawn in all, as the three repetitions of each track movement are themselves repeated 200 times.

We are ready to move the road. Due to the dual repetition previously discussed, two nested FOR . . . NEXT loops are required. The inner loop moves the road by the selected wander factor NUMLINES times, while the outer loop repeats the wander TRKLEN times. When the outer loop ends, the game is done.

The outer loop begins on line 170. TRKCNT, the number of times the wander has been executed, runs from 1 to TRKLEN, or 200 times. The outer loop ends on line 310.

Line 180 randomly selects the wander direction that is

to be repeated in the inner loop. The RND function is $z = \text{RND}(x)$ where if x is positive or omitted, the next random number is returned. If x is 0, the last number generated is repeated, and if x is negative, the sequence is reseeded, as it is when **RANDOMIZE**(x) is executed. We only need a positive value to return the next number, because we have already seeded the sequence.

RND(x) returns a value between 0 and 1. Multiplied by 3 and made an integer, this becomes a value between 0 and 2. To test this, type:

```
FOR I=1 to 20: PRINT INT(RND(100)*3): NEXT I and
press Enter
```

(Note that we used 100 because we did not wish you to attach too much weight to the value of the parameter to RND. Any positive value will function as well as 100.) Thus, line 180 randomly generates a value of -1, 0, or 1, which is what is needed for the track wander direction. (Remember that the track cannot move more than one column at a time.) This randomly generated value is stored in TRKDIR for use in the inner loop.

The inner FOR . . . NEXT loop begins on line 190 and ends on line 300. Within this loop, the selected track movement direction is executed NUMLINES (3) times. The car will also be moved each time a track section is drawn.

Line 200 performs a trial move of the track as specified by the previously generated TRKDIR. If this new location is not past the allowable margin at column 2 or column 27, then TRKPOS, the track location, is left at the new location. (Column 27 is the right margin because the track is 13 columns wide, so when the left side of the road is at column 27, the right shoulder is one column from the right edge of the screen.) If the new location is past the margin on either side, then TRKDIR is negated, reversing the road movement direction, and a move of double the new direction is executed. This double move undoes the trial move and moves one column in the new direction.

After the new track column, TRKPOS, has been set, line 210 PRINTs the new road section at the track column on

row 24. The screen scrolls up, moving the entire road up one row.

This inner loop is repeated 3 times (as specified by NUMLINES) so the road will be smoother. The outer loop repeats this process TRKLEN times to draw the entire road. Type and run the program in Listing 17-2 to generate the moving road. Observe the effect of the wander factor of 3. You might try changing the wander factor to see what effect other values have.

Listing 17-2. Racecar—Moving Track.

```

10 REM Racecar—Initial screen & moving track.
100 DEFINT A-Z:SCREEN 0,0:COLOR 7,0,0:WIDTH 40:LOCATE ,,0:
    CLS:KEY OFF
110 RANDOMIZE(VAL(RIGHT$(TIME$,2)))
120 TRKWIDTH=9:TRKPOS=15
130 NUMLINES=3:TRKLEN=200
150 FOR I=1 TO 23:LOCATE 24,TRKPOS:
    PRINT "]]" SPC(TRKWIDTH) "["[:NEXT I
170 FOR TRKCNT=1 TO TRKLEN
180   TRKDIR=1-INT(RND(100)*3)
190   FOR TRKSECTION=1 TO NUMLINES
200     TRKPOS=TRKPOS+TRKDIR:IF (TRKPOS<2) OR (TRKPOS>27) THEN
        TRKPOS=TRKPOS-2*TRKDIR:TRKDIR=-TRKDIR
210     LOCATE 24,TRKPOS:PRINT "]]" SPC(TRKWIDTH) "["[:
300   NEXT TRKSECTION
310 NEXT TRKCNT
320 END

```

Programming the Car

The program shown in Listing 17-3 adds the car to the moving road. Line 140 initializes the position of the car to the middle of the road, and initializes the car to move straight down the road.

Before the car is moved and redrawn, we first check the keyboard for a change of direction. This is done on line 220. Keyboard input, if any, is obtained with the INKEY\$ special variable. The input is then checked against z, x, and c (all lowercase) to see if CARDIR (the direction) should be set to -1 (left), 0 (straight), or 1 (right).

Line 230 sets the new location of the car based on the value stored in CARDIR. Line 230 also checks for a colli-

sion. If the SCREEN function reports that a blank (ASCII value 32) exists at the car's new location, then there is no collision, and the program goes around the collision-handling code to line 280.

If the SCREEN function does not find a blank at the car's new location, then a collision must have occurred. The program then proceeds to line 240.

Lines 240 and 250 check for collision with the right shoulder and the left shoulder, respectively. If line 240 finds the right shoulder, indicated by the left-bracket character (ASCII value 91), then the car is moved back two columns to the left. This creates the impression that the car is bouncing off the shoulder. Similarly, if line 250 finds the left shoulder, indicated by the right-bracket character (ASCII value 93), then the car is moved two columns to the right.

Note that lines 230, 240, and 250 only work because we have structured the game properly. We can assume that any nonblank character is the shoulder only because we have made sure that no other characters can be in the car's path. If we had obstacles in the road, for example, we could not make this assumption. Also, either line 240 or line 250, but not both, can be executed each time through the loop, because there is no way the car can hit both shoulders at the same time. These conditions may seem obvious, but if this game were more complex, we would have to be careful about the assumptions we make.

Line 260 which, like lines 240 and 250 only executes if a collision has occurred, makes the collision sound by executing the BEEP statement three times in a row. This produces a rather penetrating sound for about 1 second, enough time for the player to realize he has crashed, but short enough so the flow of the game is not broken.

Line 280 is executed after the program has "dealt with" any collision and the car has been moved. Line 280 draws the car on the screen on row 10 at the current car column (CARPOS). Note that the shoulder is never disturbed by collisions, because we do not draw the car until after the bounce has occurred. Also, note that we do not draw the car until after we have drawn the new road section and

scrolled the screen. This means the previous car character has been moved up one row before the new one is drawn; this is how we leave a trail behind the car. Again, the timing relationship between the road and the car is critical; if we checked in the wrong sequence, we might get collisions when it appeared none had occurred or vice versa. For example, if we drew the car before we scrolled, then it would appear to be on row 9, but the collision check would occur on row 10, where the car was drawn just before being scrolled.

Listing 17-3. Racecar—Car.

```

10 REM Racecar--Initial screen, moving track, & car.
100 DEFINT A-Z:SCREEN 0,0:COLOR 7,0,0:WIDTH 40:LOCATE ,,0:
    CLS:KEY OFF
110 RANDOMIZE(VAL(RIGHT$(TIME$,2)))
120 TRKWIDTH=9:TRKPOS=15
130 NUMLINES=3:TRKLEN=200
140 CARPOS=TRKPOS+TRKWIDTH\2:CARDIR=0
150 FOR I=1 TO 23:LOCATE 24,TRKPOS:
    PRINT "]]" SPC(TRKWIDTH) "["[:NEXT I
170 FOR TRKCNT=1 TO TRKLEN
180   TRKDIR=1-INT(RND(100)*3)
190   FOR TRKSECTION=1 TO NUMLINES
200     TRKPOS=TRKPOS+TRKDIR:IF (TRKPOS<2) OR (TRKPOS>27) THEN
        TRKPOS=TRKPOS-2*TRKDIR:TRKDIR=-TRKDIR
210     LOCATE 24,TRKPOS:PRINT "]]" SPC(TRKWIDTH) "["[:
220     K$=INKEY$:IF K$="z" THEN CARDIR=-1 ELSE IF K$="x" THEN
        CARDIR=0 ELSE IF K$="c" THEN CARDIR=1
230     CARPOS=CARPOS+CARDIR:IF SCREEN(10,CARPOS)=32 GOTO 280
240     IF SCREEN(10,CARPOS)=91 THEN CARPOS=CARPOS-2
250     IF SCREEN(10,CARPOS)=93 THEN CARPOS=CARPOS+2
260     BEEP:BEEP:BEEP
280     LOCATE 10,CARPOS:PRINT "^";
300   NEXT TRKSECTION
310 NEXT TRKCNT
320 END

```

Programming the Housekeeping

Shown in Listing 17-4 is the completed *Racecar* program. We have already set the length of the race on line 130 and the starting conditions for the car on line 140. The only other housekeeping task is handling crashes.

The number of crashes is set to zero by line 160. Line

Listing 17-4. Racecar—Finished Version.

```

10 REM Racecar—finished version.
100 DEFINT A-Z:SCREEN 0,0:COLOR 7,0,0:WIDTH 40:LOCATE ,,0:
    CLS:KEY OFF
110 RANDOMIZE(VAL(RIGHT$(TIME$,2)))
120 TRKWIDTH=9:TRKPOS=15
130 NUMLINES=3:TRKLEN=200
140 CARPOS=TRKPOS+TRKWIDTH\2:CARDIR=0
150 FOR I=1 TO 23:LOCATE 24,TRKPOS:
    PRINT "]" SPC(TRKWIDTH) "[[:NEXT I
160 NUMCRASHES=0:LOCATE 25,13:PRINT "Crashes=";:LOCATE 25,23:
    PRINT NUMCRASHES;
170 FOR TRKCNT=1 TO TRKLEN
180   TRKDIR=1-INT(RND(100)*3)
190   FOR TRKSECTION=1 TO NUMLINES
200     TRKPOS=TRKPOS+TRKDIR:IF (TRKPOS<2) OR (TRKPOS>27) THEN
        TRKPOS=TRKPOS-2*TRKDIR:TRKDIR=-TRKDIR
210     LOCATE 24,TRKPOS:PRINT "]" SPC(TRKWIDTH) "[["
220     K$=INKEY$:IF K$="z" THEN CARDIR=-1 ELSE IF K$="x" THEN
        CARDIR=0 ELSE IF K$="c" THEN CARDIR=1
230     CARPOS=CARPOS+CARDIR:IF SCREEN(10,CARPOS)=32 GOTO 280
240     IF SCREEN(10,CARPOS)=91 THEN CARPOS=CARPOS-2
250     IF SCREEN(10,CARPOS)=93 THEN CARPOS=CARPOS+2
260     BEEP:BEEP:BEEP
270     NUMCRASHES=NUMCRASHES+1
280     LOCATE 10,CARPOS:PRINT "^";
290     LOCATE 25,23:PRINT NUMCRASHES;
300   NEXT TRKSECTION
310 NEXT TRKCNT
320 FOR J=1 TO 1000:NEXT J
330 IF INKEY$<>"" THEN 320
340 LOCATE 24,7:PRINT "PRESS ANY KEY TO CONTINUE";
350 A$=INKEY$:IF A$="" THEN 350 ELSE CLS
360 END

```

160 also displays this initial number of crashes on row 25. Note that here we are taking advantage of a quirk of BASIC. Row 25 does not scroll with the rest of the screen, even after KEY OFF has been executed. We can take advantage of this by displaying the number of crashes on row 25 where it will remain throughout the game.

Line 270 adds one to the number of crashes when line 230 detects a crash. Line 290 displays the current number of crashes on row 25 each time through the loop, whether the number of crashes has changed or not. When the game is concluded, the number of crashes remains on the screen on row 25 and serves as a final score.

Note that if the PRINT statement on line 290 did not have a semicolon at the end, the screen (except for row 25) would scroll, creating gaps in the shoulders and making the game unplayable.

IMPROVING RACECAR

Type and run Listing 17-4, the finished version of *Racecar*. Try to match the play of the game to the design process we've gone through. A fully commented version of *Racecar* is shown in Listing 17-5. Study the game thoroughly before proceeding.

Listing 17-5. Racecar—Finished, Commented Version.

```

10 REM Racecar—an arcade-style game—commented version.
95 REM Make variables integers, set b/w text mode, clear screen
100 DEFINT A-Z:SCREEN 0,0:COLOR 7,0,0:WIDTH 40:LOCATE ,,0:
    CLS:KEY OFF
105 REM Seed random # generator so game is different each time
110 RANDOMIZE(VAL(RIGHT$(TIME$,2)))
115 REM Initialize game parameters
120 TRKWIDTH=9:TRKPOS=15
130 NUMLINES=3:TRKLEN=200
135 REM Start car in middle of track and heading straight
140 CARPOS=TRKPOS+TRKWIDTH\2:CARDIR=0
145 REM Draw initial section of track
150 FOR I=1 TO 23:LOCATE 24,TRKPOS:
    PRINT "]]" SPC(TRKWIDTH) "[[:NEXT I
155 REM Initialize crash count display
160 NUMCRASHES=0:LOCATE 25,13:PRINT "Crashes=";:LOCATE 25,23:
    PRINT NUMCRASHES;
165 REM Main loop for actual game play
170 FOR TRKCNT=1 TO TRKLEN
175     REM Get a random direction to move track in (-1,0 or 1)
180     TRKDIR=1-INT(RND(100)*3)
185     REM Repeat move track in direction so it wanders properly
190     FOR TRKSECTION=1 TO NUMLINES
195         REM Find new place to put track, if it would hit the
196         REM side of the screen then reverse motion
200         TRKPOS=TRKPOS+TRKDIR:IF (TRKPOS<2) OR (TRKPOS>27) THEN
            TRKPOS=TRKPOS-2*TRKDIR:TRKDIR=-TRKDIR
205         REM Put the next line of the track on the screen,
206         REM cause screen to scroll up one line
210         LOCATE 24,TRKPOS:PRINT "]]" SPC(TRKWIDTH) "[["
215         REM Set a new car direction if a direction key hit
220         K$=INKEY$:IF K$="z" THEN CARDIR=-1 ELSE IF K$="x" THEN
            CARDIR=0 ELSE IF K$="c" THEN CARDIR=1

```

Listing 17-5—cont. Racecar—Finished, Commented Version.

```

225     REM Find new car position and check for side of road
230     CARPOS=CARPOS+CARDIR:IF SCREEN(10,CARPOS)=32 GOTO 280
234     REM Car has hit side of track
235     REM If car hit right side of track, it should go left
240     IF SCREEN(10,CARPOS)=91 THEN CARPOS=CARPOS-2
245     REM If car hit left side of track it should go right
250     IF SCREEN(10,CARPOS)=93 THEN CARPOS=CARPOS+2
255     REM Beep so player knows he/she crashed
260     BEEP:BEEP:BEEP
265     REM Add to the number of crashes to be displayed
270     NUMCRASHES=NUMCRASHES+1
275     REM draw car in its new position
280     LOCATE 10,CARPOS:PRINT "^";
285     REM display the number of crashes
290     LOCATE 25,23:PRINT NUMCRASHES;
300     NEXT TRKSECTION
310 NEXT TRKCNT
315 REM Short delay to help stop key inputs
320 FOR J=1 TO 1000:NEXT J
325 REM Clear the inkey buffer
330 IF INKEY$<>"" THEN 330
340 LOCATE 24,7:PRINT "PRESS ANY KEY TO CONTINUE";
350 A$=INKEY$:IF A$="" THEN 350 ELSE CLS
360 END

```

In Chapter 11, we suggested many improvements to *Blockbuster* that are applicable to *Racecar* as well. These include saving the high score to disk, making on-screen instructions available, and pausing at the end of the game and then allowing the game to be restarted immediately.

You might want to add color to *Racecar*. Using other characters to represent the shoulders and car, in conjunction with color, could make the game visually quite attractive. Color is one of the strengths of text mode, and we suggest that you add it. In this connection, if your monitor can display text in 80 columns, you can modify the game for 80-column operation. Thus, you could use two characters where one character is used now; by combining characters, the shoulders and the car could be better drawn.

The race can be of variable length, perhaps randomly selected or selected by the player. The speed of the game can also vary. The speed can increase during the game, or can be player-controllable with a key serving as the equiv-

alent of an accelerator. Also, the road width can vary between 5 and 11 columns, for example, to add challenge. Width, speed, and length can be combined to establish skill levels which increase in difficulty as the players improve their skills.

There can be obstacles in the road that the car has to avoid (or, for that matter, collide with to gain extra points). These obstacles can be combined with wide spots in the road to create islands that the car has to pass on one side or the other. Other cars moving at different speeds would be excellent obstacles, though the programming would be difficult to implement.

One of the best additions to any game is a two-player mode. For *Racecar*, this would mean having two cars, each controlled with a different set of keys. There are a number of ways to implement two-player mode, but one good approach would be to have two identical tracks next to each other so both players would meet the same challenges at the same time.

A task you may find instructive would be programming a similar game in one of the graphics modes to gain a better understanding of the strengths and weaknesses of text and graphics modes.

Many other improvements can be made to *Racecar*. You may prefer to design your own game. *Racecar* is the last game we will design in this tutorial, so use it as a jumping-off point for your own work—and remember that you learn the most from your own efforts.

CHAPTER 18

SUMMARY OF THE TUTORIAL

You have finished reading the tutorial, and you have a practical knowledge of Advanced BASIC's graphics capabilities. Let's review what you have learned.

You first learned how to start the computer, how to create a working disk for this tutorial, and how to make a backup copy for safety. You also learned about the equipment needed for graphics, especially that there are two possible screen adapters in the PC, with the Color/Graphics Adapter necessary in order to perform graphics.

You learned that there are two primary display modes, text and graphics modes. There are two graphics modes, medium-resolution mode and high-resolution mode. Medium-resolution mode provides four colors with 320 column by 200 row resolution, while high-resolution mode is in black and white only but provides 640 column by 200 row resolution. Medium-resolution is the general-purpose graphics mode, combining color and detail, while high-resolution mode provides the best resolution available on the PC.

You learned each of the graphics commands in turn. These commands have many applications—the tutorial applied them to graphing, a pie chart, a pong-type game, and character generation. The graphics commands are powerful, versatile, and represent a major improvement over BASIC commands available on earlier microcomputers.

Text-mode graphics were described. Text-mode graphics tend to be fast and easy to use for small programs but

not well suited for large-scale graphics applications. Text-mode graphics are not very flexible; also, there are no special BASIC graphics commands for text mode as there are for graphics mode. Text-mode graphics are most useful when compatibility with the Monochrome Adapter is needed.

WHERE DO YOU GO FROM HERE?

What can you do with your new knowledge? Well, you have all the tools needed to perform virtually any graphics application that is not time-critical; even then, the GET and PUT statements often can help. You have the ability to control completely the pixel—the basic building block of graphics—with the PSET, PRESET, and POINT commands. You can also draw lines, curves, or rectangles with a single command, color areas easily with PAINT, save forms for later rapid reproduction with GET and PUT, and use command strings to define and manipulate complex shapes with DRAW. Finally, and perhaps most importantly, you have an idea of how to apply this knowledge to designing a program.

This doesn't mean that your graphics education is complete. The graphics commands are tools, and the program design you have learned is only a preliminary step. You will become adept at using the graphics commands with experience.

Graphs, games, and drawing and design packages are within your capabilities. A simple graphics package for producing colorful artwork would be exciting and achievable; three-dimensional imaging would be a challenge. An enhanced character generator, perhaps applied to multilingual education, would be a useful educational tool. Simulation animation, (the visual representation of a spacecraft's trajectory, for example) would be an impressive application of the PC's graphics.

PART 3

ADVANCED TOPICS

CHAPTER 19

A GRAB BAG OF GRAPHICS TRICKS

This chapter presents a potpourri of graphics-related “tricks” and useful facts, including several ways to manipulate colors and a means of controlling the state of the Caps Lock and Num Lock keys.

There is no particular order to the items we will present; each is useful in and of itself. Many of the items function in a rather esoteric way; we will not explain why they work but simply demonstrate their use, though the workings of some of them are explained in Chapter 20.

We use the OUT statement several times in this chapter. The OUT statement directly controls the ports that are used to communicate with the hardware in the PC. If the OUT statement is used incorrectly, it is possible, although unlikely, that the display can be damaged. Make sure you type commands *exactly* as shown when the OUT statement is used. If you have trouble, such as the display going blank or becoming garbled, immediately turn off your display and reboot the system by turning off the computer for approximately 10 seconds and then turning it back on.

Do not experiment with the OUT statement unless you know what you are doing. Read the *Technical Reference* manual before you use the OUT statement except as it is used in this book.

One other note is that we use the DEF SEG statement often in this chapter. When the DEF SEG statement is

used, type it *as shown* or the function following the `DEF SEG` will not perform properly. For example, when we demonstrate saving the screen to disk, a `DEF SEG = &HB800` statement must be performed before the screen is saved, or the save will not work properly. We will describe the `DEF SEG` statement in detail in the next chapter.

COLOR IN HIGH-RESOLUTION MODE

There are two ways to get color in high-resolution graphics mode. One works only on RGB monitors, while the other works only on television sets and composite monitors. The two methods produce results that differ sharply.

When using an RGB monitor in high-resolution mode, you may select any one, but only one at a time, of the 16 colors available on the PC for the foreground color (color 1). To select a color, type the BASIC command `OUT &H3D9,color` and press Enter where **color** is a value between 0 and 15 as shown in Table 19-1. For example, select high-resolution mode by typing:

`SCREEN 2` and press Enter

and type:

`OUT &H3D9,4` and press Enter

and all the white pixels on the screen are red. (Remember, this will only work on an RGB monitor.) Type:

`OUT &H3D9,9` and press Enter

and the pixels are light blue. As with switching palettes in medium-resolution mode, every pixel on the screen immediately changes color. The program in Listing 19-1 shows each of the colors available in high-resolution mode on an RGB monitor. Color 8 is a gray that may not show up well unless your display brightness is high.

ARTIFACTING

The method described in the preceding section does not produce color on television sets and composite monitors.

Table 19-1. High-Resolution Graphics Mode Foreground Colors on RGB Monitors

NUMBER	COLOR*	NUMBER	COLOR*
0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	High-Intensity White

Listing 19-1. High-Resolution Graphics Mode Color on RGB Monitors.

```

100 REM Program to demonstrate color in high-resolution
110 REM  graphics mode on RGB monitors only.
120 SCREEN 2:KEY OFF:CLS      'Set screen
130 REM We will make the circle each color in turn
140 CIRCLE (320,100),50:PAINT STEP(0,0)
150 LOCATE 18,34:PRINT "COLORED CIRCLE" 'Label it
160 REM Do all 15 colors but black
170 FOR I=1 TO 15
180   OUT &H3D9,I             'Select color
190   FOR J=1 TO 1000:NEXT J   'Delay for viewing
200 NEXT I
210 REM Wait to end
215 LOCATE 21,28:PRINT "PRESS ANY KEY TO CONTINUE"
220 A$=INKEY$:IF A$="" THEN 220
230 SCREEN 0,1:WIDTH 40       'Reset screen
240 END

```

However, a second method, called *artifacting*, can be used to produce a wide variety of colors in high-resolution mode on composite displays (including tvs) only.

We are going to describe and demonstrate artifacting; a technical discussion is beyond the scope of this book. Artifacting on the PC is the process of putting high-resolution white pixels on the screen in patterns that produce color effects on displays driven with a composite signal. The resolution available when using artifacting on the PC is approximately 160 columns by 200 rows in 16 colors

(see Fig. 14 in the color photograph section). Additional colors can be produced, but resolution diminishes further because the additional colors are produced by combining several dots of differing colors. Artifacting works best with large areas of solid color. The results achieved with artifacting can be excellent, but we must emphasize that on RGB monitors, artifacting only produces various patterns of white dots.

On the PC, artifacting is performed in high-resolution mode. Composite monitors cannot display all of the 640 pixels across the screen that would be displayed by an RGB monitor; instead, each four-pixel group on any row produces a single dot of one color. In effect, every four pixels define what we might call a “mega-pixel,” which is the dot displayed on the composite screen. The four-pixel groups start with a pixel in a column evenly divisible by 4, so that four-pixel groups start in columns 0, 4, 8, 12, and so on, with 160 four-pixel groups on each of the 200 rows.

A demonstration follows. Type and run the program shown in Listing 19-2. (If you have only an RGB monitor, this program will not produce the desired effect.) Note the many bright colors. (The actual colors you see depend on your display. Some television sets have automatic tuning circuitry that can change colors or make some colors appear as white.)

The statement **OUT &H3D8,26** on line 150 must always be executed to turn on color before artifacting is performed. The **FOR . . . NEXT** loop from lines 180 through 350 draws and labels a solid block of each of the 16 colors in turn, beginning with black and ending with white.

The key line in this program is 300, which draws each of the vertical lines that comprise the block of solid color. The **FOR . . . NEXT** loop starting at line 270 varies *L* to select each of the four low-order bits of *I*, the variable which selects the color currently being used. These four bits in turn determine the white or black status of the corresponding four pixels controlling the “mega-pixel” being drawn. The four pixels can be arranged in 16 different ways, producing 16 different colors. When all four pixels are white, the color of the “mega-pixel” will be white, and

Listing 19-2. Color Generation by Artifacting.

```

100 REM Program to demonstrate color generation in
110 REM hi-res mode by artifacting.
120 REM Set screen to hi-res mode
130 SCREEN 2:KEY OFF:CLS
140 REM Enable color burst so TV will display color
150 OUT &H3D8,26
160 REM Label column that color numbers will go in
170 LOCATE 1,38:PRINT "COLOR #"
180 REM Draw using each of the 16 colors in turn
190 FOR I=0 TO 15
200 REM Set point at which to begin drawing colored area
210 PRESET(120,I*8+16)
220 REM Draw columns 100 to 250
230 FOR K=100 TO 250 STEP 4
240 REM Draw set of 4 pixels, producing one
250 REM artifacted colored pixel. Color is based
260 REM on the 4-bit pattern in I
270 FOR L=3 TO 0 STEP -1
280 REM Draw one column in white, if bit is 1, or
290 REM in black, if bit is 0
300 LINE -STEP(0,7),SGN(I AND 2^L):PRESET STEP(1,-7)
310 NEXT L
320 NEXT K
330 REM Label color number
340 LOCATE I+3,40:PRINT I
350 NEXT I
360 LOCATE 24,27:PRINT "PRESS ANY KEY TO CONTINUE";
370 AS=INKEY$:IF AS="" THEN 370
380 SCREEN 0,1:WIDTH 40 'Reset screen
390 END

```

when all four are black, black is produced. Run the program to see the 14 other colors.

If the preceding discussion (especially the term "bit") confuses you, don't despair, as the material covered in the next chapter may help. Artifacting is not complicated to use, but it is difficult to *explain* in nontechnical terms. In any case, the best way to learn about artifacting is to experiment with it.

Artifacting is worth learning. On many computers, it is the primary way of producing color, and on the PC, it gives the best combination of color and resolution available. There are two drawbacks to artifacting: (1) the figure lack detail, and (2) artifacting does not work on RGB monitors.

MORE ON COLOR

In text mode, BASIC allows us to have only the 8 low-intensity colors for the background parameter. It is possible to get all 16 background colors, but in doing so, the computer cannot produce blinking characters.

To enable all 16 background colors, type:

```
OUT &H3D8,n and press Enter
```

where *n* is 8 when the screen width is 40 columns, and *n* is 9 when the screen width is 80 columns. Then, to produce a high-intensity background, add 16 to the value of the foreground color when you use the COLOR statement. The important point here is that the value of the foreground color affects the background. For example, select 40-column color text mode and type:

```
WIDTH 40: OUT &H3D8,8 and press Enter
```

to disable blink and enable the 16 background colors. Now type:

```
COLOR 7,1: PRINT "DARK BLUE" and press Enter
COLOR 7 16,1: PRINT "LIGHT BLUE" and press Enter
```

to produce both the normal and high-intensity backgrounds.

However, if we type:

```
COLOR 23: PRINT "NOT BLINKING" and press Enter
```

and we have lost the ability to produce blinking characters.

To restore blinking and turn off high-intensity background colors, type:

```
OUT &H3D8,n and press Enter
```

where *n* is 40 in 40-column mode and 41 in 80-column mode. If we now type:

```
OUT &H3D8,40 and press Enter
```

the characters that have high-intensity backgrounds will begin to blink.

The interpretation of the attribute byte returned by the `SCREEN` function (described in Chapter 16) differs slightly when 16 background colors are turned on and the blink is turned off. The foreground and background colors are determined the same way, except you can tell if the background color is a high-intensity color by a statement such as `IF SCREEN(ROW,COL,1) AND 128 THEN PRINT "HIGH-INTENSITY BACKGROUND"`.

This is the same test used for blinking. Because there is no blinking when there are 16 background colors, there is no conflict.

COLORED TEXT IN GRAPHICS MODE

In medium-resolution graphics mode, BASIC is able to `PRINT` text in color 3 (white or brown) only, even though there are two other colors in each palette. We can select any of the palette colors as the `PRINT` color with the statement `DEF SEG: POKE &H4E,color` where `color` is 1, 2, or 3 for the desired color from the current palette. If you select color 0, the background color, the characters will be invisible, because the foreground and background colors will be the same.

For example, in medium-resolution mode, with palette 1 selected, type:

```
DEF SEG: POKE &H4E,2: PRINT "MAGENTA TEXT" and  
press Enter  
POKE &H4E,1: PRINT "CYAN TEXT" and press Enter  
POKE &H4E,3: PRINT "WHITE TEXT" and press Enter
```

It is only necessary to execute the `DEF SEG` statement once at the beginning of a program, or after a different `DEF SEG` has been executed.

The use of a method, such as this one, which involves modifying undocumented sections of BASIC involves some risk. The technique may not work with a different version of BASIC or with Advanced BASIC on another machine. By way of contrast, the `PSET` statement will work in any version of Advanced BASIC on any computer that supports PC-DOS. IBM has, by printing a manual,

committed itself to supporting and maintaining the documented features of BASIC, but we have no such guarantee with the text-color modification just discussed. Such tricks are often useful when a necessary function is omitted, but should be avoided otherwise.

THE THIRD PALETTE

As we hinted previously, there is a third, unofficial palette available on the PC. Colors 1 and 3 are cyan and white, as with palette 1, but color 2 is a vivid red. In fact, this third palette might have the most eye-catching color set of the three palettes (see Fig. 15 in the color photograph section). The drawback is that the palette can be displayed only on an RGB monitor.

The third palette is produced by disabling the burst in medium-resolution mode. This produces a black-and-white display on television sets and composite screens, but on RGB monitors the third palette is produced instead. For example, type:

```
SCREEN 1,1 and press Enter
```

to select medium-resolution mode with the burst turned off. Clear the screen, and type:

```
LINE (10,100)-(90,190),1,BF and press Enter
LINE (110,100)-(190,190),2,BF and press Enter
LINE (210,100)-(290,190),3,BF and press Enter
```

to display the three colors available.

When you switch from palette 0 to palette 1, the screen does not clear. However, when you switch from palettes 0 or 1 to the unofficial palette, or vice versa, the screen does clear.

The program in Listing 19-3 shows each of the three palettes. Lines 160-280 select each of the three palettes in turn and call the subroutine at line 310 to draw three circles in colors 1, 2, and 3 of that palette. The third palette will appear in black and white on composite displays.

Listing 19-3. Three Color Palettes.

```

100 REM Program to demonstrate the three color
110 REM   palettes available on the IBM PC.
120 REM   The last palette does not work on
130 REM   TV's.
140 SCREEN 1,0:CLS:KEY OFF      'Set screen
150 COLOR 0                    'Black background
160 REM Do palette 0 first
170 COLOR ,0                    'Select palette 0
180 REM Set label & label location
190 LBL$="Palette 0":COL=17
200 GOSUB 310                    'Draw circles in colors 1,2,3
210 REM Palette 1
220 COLOR ,1                    'Select palette 1
230 LBL$="Palette 1":COL=17     'Set label/loc
240 GOSUB 310                    'Circles in three colors
250 REM Unofficial palette
260 SCREEN ,1                    'Disable color burst
270 LBL$="Unofficial Palette":COL=12
280 GOSUB 310                    '3 circles
290 SCREEN 0,1:WIDTH 40        'Reset screen
300 END
310 REM Subroutine to draw circles in
320 REM   colors 1, 2, and 3. Screen is
330 REM   labelled with sting in LBL$
340 CLS
350 CIRCLE(80,100),30,1:PAINT STEP(0,0),1
360 CIRCLE(160,100),30,2:PAINT STEP(0,0),2
370 CIRCLE(240,100),30,3:PAINT STEP(0,0),3
380 REM Label screen
390 LOCATE 21,COL:PRINT LBL$
400 LOCATE 24,8:PRINT "PRESS ANY KEY TO CONTINUE";
410 A$=INKEY$:IF A$="" THEN 410
420 RETURN

```

CHECKING THE SCREENS

An IBM PC can have a Color/Graphics Adapter, a Monochrome Adapter, or both. It is often essential that you know what screens are available, which one BASIC is currently using, and how to switch between the screens.

The program *Switch*, discussed in Chapter 2, provides you with a means to go from one screen to the other. An important point is that when you switch from one screen to the other, you do not necessarily have to clear the screen. If you do not clear the screen, the information dis-

played on the old screen remains. If, *and only if*, you have a two-adapter system, run the program in Listing 19-4 for an example of this. Lines 180-250 switch to the Color/Graphics Adapter, lines 260-310 draw on the graphics screen, and lines 320-360 then switch back to the monochrome screen. The graphics display remains on the color monitor even though the cursor and all subsequent text are back on the monochrome screen.

When you select a new display, however, that screen is automatically cleared. In short, you can put information on a screen, then switch to the other screen without erasing the information, but when you switch back to the first screen, the first screen will be erased.

Listing 19-4. Two-Adapter Graphics.

```

100 REM Program to switch to color/graphics adapter,
110 REM   draw graphics, then return to the monochrome
120 REM   screen with the graphics intact.
130 REM   May be started on either screen--always ends
140 REM   on the monochrome screen.
150 REM *
160 REM * Will work *only* with two-adapter systems!!!
170 REM *
180 DEF SEG=0 'Point segment to BIOS variables
190 RFM Switch to color/graphics adapter
200 REM First clear old screen before switching
210 KEY OFF:CLS
220 A=PEEK(1040):POKE 1040,(A AND 207) OR 32
230 REM Set medium-resolution graphics screen
240 REM   with palette 0
250 SCREEN 1,0:COLOR 0,2:CLS:KEY OFF
260 REM Draw circles in three colors
270 CIRCLE(80,100),80,3:PAINT STEP(0,0),3
280 CIRCLE(160,100),80,2:PAINT STEP(0,0),2
290 CIRCLE(240,100),80,1:PAINT STEP(0,0),1
300 REM Label the artwork
310 LOCATE 24,9:PRINT "Circles in graphics mode";
320 REM Switch to monochrome adapter
330 A=PEEK(1040):POKE 1040,A OR 48
340 REM Set monochrome screen mode & cursor
350 SCREEN 0,0:COLOR 7,0:LOCATE 1,1,1,12,13
360 KEY ON:WIDTH 80
370 END

```

To tell if there is a Color/Graphics Adapter in a PC, use two statements that take the form **DEF SEG=&HB800: A=PEEK(0): POKE 0,(A+1) MOD 256**, then **IF PEEK(0) < > (A+1) MOD 256 THEN PRINT "NO COLOR/GRAPHICS ADAPTER"** and press Enter. Here the DEF SEG statement sets BASIC to look at that part of memory where the Color/Graphics Adapter resides, and the PEEK and POKE statements check whether there is anything there. We will discuss PEEK, POKE, and DEF SEG in detail in the next chapter.

To check for the Monochrome Adapter, two statements such as **DEF SEG=&HB000: A=PEEK(0): POKE 0,(A+1) MOD 256** and press Enter, then **IF PEEK(0) < > (A+1) MOD 256 THEN PRINT "NO MONOCHROME ADAPTER"** and press Enter.

The drawback to this method is that the contents of the upper-left corner of the screen will be changed. If this is a problem, execute the statement **POKE 0,A** as the next line after the **IF . . . THEN** statement, which tests for the presence of the adapter, to restore the original character.

The program in Listing 19-5 looks for the presence of each adapter. Lines 150-180 check for the presence of the Color/Graphics Adapter, with the **IF . . . THEN . . . ELSE** statement on line 180 being **TRUE** if there is a Color/Graphics Adapter and **FALSE** otherwise. The **MOD 256** portion of line 170 is necessary to make sure the value is less than or equal to 255, because attempting to **POKE** a value greater than 255 causes an error. (We will describe the **POKE** statement in the next chapter.)

Similarly, lines 210-240 check for the presence of the Monochrome Adapter, with the **IF . . . THEN . . . ELSE** statement on line 240 being **TRUE** if there is a Monochrome Adapter and **FALSE** otherwise.

When you run the program in Listing 19-5, note the change in the upper-left corner of the screen. In text mode, an exclamation point or the letter P will often appear. As previously explained, the program changes the contents of this area of the display; this condition can be remedied by restoring the original value, as stored in the variable A, after the test is finished.

Listing 19-5. Determine Display Adapter Installed.

```

100 REM Program to determine which display
110 REM adapters are installed in PC.
120 CLS:KEY OFF
130 REM Check for color/graphics adapter
140 REM Point segment to graphics screen
150 DEF SEG=&HB8000
160 LOCATE 10,10:PRINT "COLOR/GRAPHICS ADAPTER: ";
170 A=PEEK(0):POKE 0,(A+1) MOD 256
180 IF PEEK(0)<>(A+1) MOD 256 THEN PRINT "NO" ELSE
PRINT "YES"
190 REM Check for monochrome adapter
200 REM Point segment to monochrome screen
210 DEF SEG=&HB0000
220 LOCATE 15,10:PRINT "MONOCHROME ADAPTER: ";
230 A=PEEK(0):POKE 0,(A+1) MOD 256
240 IF PEEK(0)<>(A+1) MOD 256 THEN PRINT "NO" ELSE
PRINT "YES"
250 LOCATE 20,10:PRINT "PRESS ANY KEY TO CONTINUE";
260 A$=INKEY$:IF A$="" THEN 260
270 SCREEN 0,1 :CLS 'Reset screen
280 END

```

BASIC will perform operations on only one screen at a time. The program *Switch* allows us to select which screen to use, but we do not necessarily know which screen BASIC is using at any given time. For example, a PC could start up on either the monochrome or graphics screen. For a graphics program, we would like to test which screen we are on: when it is the monochrome screen, we would test for the Color/Graphics Adapter as previously described. If a Color/Graphics Adapter exists, we would switch screens, and, if not, we would print a message indicating that our program could not run on this system.

We can tell which adapter is selected with a statements: **DEF SEG=0** and press Enter, then **IF PEEK(&H410) AND &H30 <> &H30 THEN PRINT "COLOR/GRAPHICS ADAPTER"** and press Enter.

The program shown in Listing 19-6 switches the display to the Color/Graphics Adapter if one exists, or notifies the user if there is no Color/Graphics Adapter. First, the program checks which screen BASIC is using. If the screen is the Color/Graphics Adapter, the program ends. If the

monochrome screen is in use, the program checks for the Color/Graphics Adapter. If the Color/Graphics is present, then the program switches to that screen; otherwise, the user is notified that there is no Color/Graphics Adapter available. These program lines can be used as a part of any of your programs that require a Color/Graphics Adapter to run.

Listing 19-6. Switch with Error Message.

```

100 REM Program to switch display to color/graphics adapter
110 REM   if one exists. If not, program will notify user
120 REM   that the graphics screen does not exist.
130 REM
140 REM First, check if display is already on color/
150 REM   adapter. If so, then we're done
160 DEF SEG=0 'Point segment to BIOS variables
170 REM Check current screen
180 IF PEEK(1040) AND 48<>48 THEN GOTO 390
190 REM Display is not on color/graphics adapter. Check if
200 REM   color/graphics adapter exists.
210 DEF SEG=&HB800 'Point segment to graphics screen
220 REM Modify first memory location in screen
230 A=PEEK(0):POKE 0,(A+1) MOD 256
240 REM If it stays modified, then adapter exists
250 IF PEEK(0)=(A+1) MOD 256 THEN 320
260 REM Notify that color/graphics adapter doesn't exist
270 CLS:KEY OFF:LOCATE 10,20
280 PRINT "Color/graphics adapter does not exist!"
290 LOCATE 21,27:PRINT "PRESS ANY KEY TO CONTINUE";
300 A$=INKEY$:IF A$="" THEN 300 ELSE CLS
310 END
320 REM Color/graphics adapter exists-switch display from
330 REM   the monochrome screen to the graphics screen
340 REM Clear the monochrome screen
350 KEY OFF:CLS
360 DEF SEG=0:A=PEEK(1040):POKE 1040,(A AND 207) OR 32
370 SCREEN 0:LOCATE 1,1,1,6,7
380 KEY ON:WIDTH 40
390 REM *
400 REM * Program continues here if display is
410 REM *   successfully on the color/graphics adapter
420 REM *
430 END

```


SAVING THE SCREEN TO DISK—BSAVE AND BLOAD

All the information displayed on the screen can be saved to a disk file at any time and restored at a later time with the **BSAVE** and **BLOAD** statements. The exact form of **BSAVE** depends on the screen mode selected.

To save the information displayed on the screen in medium- or high-resolution graphics mode, type:

```
DEF SEG=&HB800: BSAVE filename,0,&H4000 and press
Enter
```

where **filename** is a string constant or variable containing any valid filename, as described in the *BASIC* manual. After this command is executed, the screen will be stored in a 16K file **filename**. There must always be at least 16K of space available on the disk when saving a graphics mode screen. To save the screen in text mode, type:

```
DEF SEG=&HB800: BSAVE filename,0,scrlen and press
Enter
```

In this example, **scrlen** is **&H1000** in 80-column text mode, and **BSAVE** creates the 4K file named **filename**. In 40-column mode, **scrlen** is **&H0800**, and **BSAVE** creates a 2K file.

To save the monochrome screen, type:

```
DEF SEG=&HB000: BSAVE filename,0,&H1000 and press
Enter
```

This creates the 4K file **filename**. Note that the monochrome screen requires the statement **DEF SEG = &HB000** while the Color/Graphics Adapter requires **DEF SEG = &HB800**.

It is possible to save any number of screens, although each must have a different filename.

For example, clear the screen in medium-resolution mode and draw two concentric circles with the command line:

```
CIRCLE(160,100),50,2: CIRCLE(160,100),30,1 and
press Enter
```

Save the screen to disk by typing:

```
DEF SEG=&HB800: BSAVE  
"CIRCLE.SCR",0,&H4000 and press Enter
```

The 16K file CIRCLE.SCR is created.

Once a file has been created with BSAVE, the saved screen may be restored within seconds with the BLOAD statement. For example, clear the screen in medium-resolution mode and type:

```
DEF SEG=&HB800: BLOAD "CIRCLE.SCR" and press Enter
```

The screen we previously saved is restored.

In general, a saved graphics screen is restored with **DEF SEG=&HB800: BLOAD filename** where **filename** is the name of a file previously created with the BSAVE statement. The screen must be in the same mode (text, medium-resolution, or high-resolution) that it was in when the file was created, or unpredictable results can occur. In the graphics modes, the screen takes several seconds to be restored with BLOAD because of the time needed to read 16K of information from the disk.

To restore the monochrome screen, type the command:

```
DEF SEG=&HB000: BLOAD filename and press Enter
```

Run the program in Listing 19-7 for a demonstration of saving and restoring a screen display via the BSAVE and BLOAD statements. Lines 120-180 create a screen image, lines 190-210 BSAVE the image to a disk file, line 230 clears the screen, and lines 240-260 restore the image from disk to screen with the BLOAD statement. The DEF SEG=&HB800 statement on line 200 must be executed before an image can be saved from or restored to the graphics screen.

Saving the screen can be useful for slide-type presentations. The displays can be created beforehand and shown in rapid succession. This avoids the tedium of waiting through the drawing of a screen (remember how slow the CIRCLE statement is), and eliminates the possibility of bugs in a drawing program marring the presentation. Incidentally, BSAVE and BLOAD are also useful for saving

Listing 19-7. BSAVE and BLOAD Statements in Medium-Resolution Mode.

```

100 REM Program to demonstrate the BSAVE and BLOAD
110 REM statements in med-res graphics mode.
120 SCREEN 1,0:COLOR 0,1:CLS:KEY OFF 'Set screen
130 REM Loop to draw concentric circles
140 FOR I=90 TO 10 STEP -10
150 CIRCLE(160,100),I
160 REM Draw band with cyan or magenta
170 PAINT STEP(0,0),(I\10 MOD 2)+1,3
180 NEXT I
190 REM Save to disk file "circles"
200 DEF SEG=&HB800 'Point segment to color screen
210 BSAVE "circles",0,&H4000 'Save 16K file
220 REM Now clear the screen and reload old screen
230 CLS
240 LOCATE 10,15:PRINT "RELOADING..."
250 FOR I=1 TO 1000:NEXT I:CLS
260 BLOAD "circles",0 'Load display back in
270 LOCATE 24,9:PRINT "PRESS ANY KEY TO CONTINUE";
280 A$=INKEY$:IF A$="" THEN 280 ELSE CLS
290 END

```

and loading machine-language subroutines to be used from BASIC programs.

A note regarding the filename parameter to BSAVE and BLOAD: if you omit the file extension (the period followed by three characters), BASIC uses the extension .BAS, as it does for a BASIC program. The command **BSAVE "TEST",0,&H4000** creates the file TEST.BAS, rather than the file TEST as you might expect. Similarly, **BLOAD "TEST",0** causes the computer to try to load the file TEST.BAS. If you want to use the name TEST with no extension, you have to use **BSAVE "TEST.",0,&H4000** and **BLOAD "TEST.",0** where the period is included in the file specification.

The BASIC manual is a good resource if you have any questions about the BSAVE and BLOAD statements. We will describe BSAVE and BLOAD in more detail in the next chapter.

PRINTING THE SCREEN

Screen displays are certainly useful, but often a hard copy of the screen is needed. Sometimes photographs of the screen will suffice, but this takes time and skill. An alternative is "dumping" the screen to your printer.

A text-mode screen can be dumped immediately by holding down the Shift key and pressing the PrtSc key, which is located just below the Enter key. This will work with any printer.

There is no built-in way to print a graphics screen, but there are a number of software products available that allow you to print a graphics screen on an IBM dot-matrix printer. There are also products that allow a medium-resolution screen to be printed in color on a Data Products P Series™ color printer. If you need a hard copy of the graphics screen, these screen-dump programs are often your best bet.

THE KEYBOARD

The keyboard is not directly related to graphics; however, proper control of keyboard input is essential to good game design and is useful in almost any software package. For example, if a program is written for uppercase input only, it is better to permanently "shift" the keyboard in software than to require the user to make sure of the shift state. Our main purpose here is to ensure that you are able to handle some of the PC's peculiarities.

For example, there is no way for the user to tell the state of either of the Caps Lock or Num Lock keys. Most computers have a small red light that indicates when the keys are set, but the PC does not. The PC user could, for example, accidentally press the Num Lock key and thus, for example, end up typing the number 7 instead of the Home key.

It is possible to check and to set the state of the Num Lock and Caps Lock keys. To see if the Caps Lock key has

P Series is a trademark of Data Products Corporation.

been toggled on (so that letters are uppercase), use an IF . . . THEN statement such as **DEF SEG=0: IF PEEK(&H417) AND &H40 THEN PRINT "CAPS LOCK ON" ELSE PRINT "CAPS LOCK OFF"**. Similarly, the state of Num Lock can be checked with **DEF SEG=0: IF PEEK(&H417) AND &H20 THEN PRINT "NUM LOCK ON" ELSE PRINT "NUM LOCK OFF"**. (When the Num Lock key is "On," the keypad produces numeric output on the screen.) Try these with the Caps Lock and Num Lock keys on and off.

The program in Listing 19-8 monitors, from BASIC, the Caps Lock and Num Lock states. The program repeatedly loops to check the state of the keys via the PEEK statement, and updates the information on the display screen. The key lines are line 230, which checks the Caps Lock state, and line 250, which checks the Num Lock state. Toggle the Caps Lock and Num Lock keys and watch the program register the changes. The DEF SEG=0 statement on line 180 must be executed before the state of the keys can be checked.

Listing 19-8. Monitor Caps Lock and Num Lock.

```

100 REM Program to monitor the states of the
110 REM   CapsLock and NumLock keys.
120 SCREEN 1,0:COLOR 0,1:CLS:KEY OFF
130 REM Print labels
140 LOCATE 3,10:PRINT "CAPSLOCK STATE"
150 LOCATE 8,10:PRINT "NUMLOCK STATE"
160 LOCATE 15,8:PRINT "PRESS THE = KEY TO END"
170 REM Point segment to BIOS variables
180 DEF SEG=0
190 REM Loop to keep checking until the
200 REM   = key hit
210 IF INKEY$="=" THEN CLS:END
220   REM CapsLock state
230   LOCATE 5,15:IF PEEK(&H417) AND &H40 THEN
      PRINT "UPPERCASE" ELSE PRINT "LOWERCASE"
240   REM NumLock state
250   LOCATE 10,15:IF PEEK(&H417) AND &H20 THEN
      PRINT "NUMERIC  " ELSE PRINT "NONNUMERIC"
260 GOTO 210

```

The Caps Lock state is set to uppercase with **DEF SEG=0: POKE &H417,PEEK(&H417) OR &H40** and

set to lowercase with **DEF SEG=0: POKE &H417, PEEK(&H417) AND (NOT &H40).**

These commands set or reset the Caps Lock key regardless of its current state. Of course, the user could change the state again, so you should frequently select the desired Caps Lock state in programs you write that control the Caps Lock state. The same is true for the Num Lock state.

The NumLock state is set to numeric with **DEF SEG=0: POKE &H417, PEEK(&H417) OR &H20** and set to nonnumeric with **DEF SEG=0: POKE &H417, PEEK(&H417) AND (NOT &H20).**

The program in Listing 19-9 keeps the Caps Lock state off (lowercase) and the Num Lock state on (numeric). Enter and run it to see that you cannot type uppercase letters or cursor-control keys by using the Caps Lock or Num Lock keys. Line 230 forces the Caps Lock state to lowercase, and line 250 forces the Num Lock state to numeric. This is done each time through the loop that checks the keyboard for characters (lines 210-300), so that if you toggle either key, it will soon be reset.

The **DEF SEG=0** statement on line 160 must be executed before the state of the Caps Lock or Num Lock keys can be set.

CLEARING EXCESS KEYS

The PC allows users to type ahead, storing the extra characters for later use in what is called a *buffer*. At some point, you have probably noticed that only a certain number of characters can be stored when typing ahead, and, if this number is exceeded, the PC emits a piercing squeal and ignores the extra characters. Fifteen characters can be stored in the buffer, and it is easy to think of a situation where a user could type more than 15 characters while the program performed some other task. When such an occurrence is likely, the program should clear the buffer of excess characters before looking for new input; otherwise, old characters typed by mistake may be incorrectly used as current input. For example, in those cases where **INKEY\$** is used to wait for a key press before proceeding,

Listing 19-9. Control Caps Lock and Num Lock.

```

100 REM Program to demonstrate controlling the
110 REM   CapsLock and NumLock states. CapsLock
120 REM   is kept LOWERCASE, and NumLock is kept
130 REM   NUMERIC.
140 SCREEN 1,0:COLOR 0,1:CLS:KEY OFF
150 LOCATE ,,1           'Set cursor on
160 DEF SEG=0            'Point segment to BIOS variables
170 REM Initial prompts
180 PRINT "TYPE AT THE KEYBOARD, AND TRY TO CHANGE"
190 PRINT "   THE CAPSLOCK AND NUMLOCK STATES."
200 PRINT "       PRESS THE = KEY TO EXIT.":PRINT
210 REM Loop to input characters & set key states
220   REM Set CapsLock lowercase
230   POKE &H417,PEEK(&H417) AND (NOT &H40)
240   REM Set NumLock numeric
250   POKE &H417,PEEK(&H417) OR &H20
260   A$=INKEY$           'Get character, if any
270   REM Check for character, end if =
280   IF A$="" THEN 230 ELSE IF A$="=" THEN 310
290   PRINT A$;           'Echo character to keyboard
300 GOTO 210
310 REM Turn NumLock back to nonnumeric to end
320 POKE &H417,PEEK(&H417) AND (NOT &H20)
330 CLS:END

```

the program should clear the buffer before waiting for a key press, so that a key pressed at some earlier time but never read does not cause the program to proceed.

As long as the excess characters are to be ignored, it is easy to dispose of them. One method, recommended in Appendix I of the *BASIC* manual, is to execute the command **DEF SEG=0: POKE 1050,PEEK(1052)**. This single command empties the buffer. The one drawback is that this is a command specific to the version of BASIC used and to the IBM PC. In general, such methods should be avoided when possible. An alternative method is a program line such as **100 IF INKEY\$ < > "" THEN 100** which obtains keyboard strokes, one at a time, until the buffer is empty. For example, enter NEW to clear memory, type the program in Listing 19-10, and RUN the program. Notice that it is difficult to get the PC to beep, even by typing fast, because on line 180, INKEY\$ is used to clear all pending characters. It is possible to produce a beep by

very rapid typing, but the object is to prevent the user from being distracted or confused during reasonable operation of the machine.

This method will work with any version of Advanced BASIC on any machine.

Listing 19-10. Key Buffer Clearing with INKEY\$.

```
100 REM Program to demonstrate the use of INKEY$
110 REM to clear the key buffer
120 REM Print 100 numbers, clearing buffer
130 REM in between PRINTS
140 SCREEN 1,0:COLOR 0,1:CLS:KEY OFF
150 FOR I=1 TO 100
160 PRINT I
170 REM Read characters out of buffer until empty
180 IF INKEY$<>"" THEN 180
190 NEXT I
200 LOCATE 24,8:PRINT "PRESS ANY KEY TO CONTINUE";
210 A$=INKEY$:IF A$="" THEN 210 ELSE CLS
220 END
```

ENTERING ALL 255 CHARACTERS

As described previously, there are 255 characters on the PC but not 255 keys on the keyboard. Use the following procedure to type any of the 255 characters. Hold down the Alt key as you type all three digits of the desired character's ASCII code (as shown in Appendix A) on the numeric keypad. Release the Alt key. The character will be produced as if you had pressed a key for it. For example, while holding down the Alt key, press the 0, 6, and 5 keys on the numeric keypad. Release the Alt key, and the character A will appear. If you look at Appendix A, you will see that 65 is the ASCII value of A. This method can be used, if necessary, to type any of the PC's 255 characters.

For instance, type:

```
SCREEN 0 and press Enter
PRINT "
```


but do *not* press Enter. (We selected text mode because only the text-mode screen can produce characters 128-255.) Now hold down the Alt key and press 2, 5, and 1 on the numeric keypad. Release the Alt key. A square-root symbol is produced, and will function like any other character. Type a closing double quote and press the Enter key. The square-root symbol is displayed.

The characters with ASCII values 0 through 31, as well as 127, cannot be produced while typing in immediate mode by the Alt method, although some of these characters produce special results. For example, Alt 7 produces a beep. The INPUT statement also fails to recognize these 33 characters. All Alt-generated characters are recognized by INKEY\$ so they can be typed into a program in response to a command such as **A\$=INKEY\$**.

There are also some situations in which BASIC becomes "confused" by the presence of some of these 255 characters in a program. Because the same result can usually be produced with the CHR\$(x) function, we suggest you avoid the Alt method of typing characters unless absolutely necessary. For example, the square-root sign could have been produced with **PRINT CHR\$(251)** instead of using the Alt key.

THE FULL PC CHARACTER SET

BASIC provides no easy way to print the special-function characters, such as character 7, the beep. However, the PC can print a full 255-character set, as shown in Appendix B. We must use a machine-language subroutine to print the characters unavailable from BASIC.

Several chapters are required to introduce you properly to the use of machine language from BASIC. What we will do is present the BASIC program lines required to produce any of the 255 characters in any available color.

The program shown in Listing 19-11 prints each of the 255 characters. Type and run it, noting that all 255 characters, including characters 7-13 and 28-31, appear. Type the DATA statements carefully, and double-check your entries.

Listing 19-11. Print 255 PC Characters.

```

100 REM Program to demonstrate printing all 255
110 REM   PC characters.
120 SCREEN 0,1:WIDTH 40:LOCATE , ,0:COLOR 7,0,0:
    CLS:KEY OFF      'Set screen
130 DEF SEG          'Point segment to BASIC interpreter
140 REM String SUBRT$ will hold machine-language routine
150 SUBRT$=""
160 REM Loop to load machine-language routine
170 REM   into SUBRT$
180 FOR J=1 TO 26:READ I:SUBRT$=SUBRT$+CHR$(I):NEXT J
190 DATA &h55,&h8B,&hEC,&h8B,&h5E,&h08,&h8A,&h07,&h8B
200 DATA &h5E,&h06,&h8A,&h1F,&hB4,&h09,&h28,&hFF,&hB9
210 DATA &h01,&h00,&hCD,&h10,&h5D,&hCA,&h04,&h00
220 COLR%=7          'Print characters in white
230 A=VARPTR(SUBRT$) 'Get SUBRT$ descriptor address
240 REM Get address of string contents (where machine-
250 REM   language routine is stored) from descriptor
260 SPECPRINT=PEEK(A+1)+PEEK(A+2)*256
270 REM Loop to call routine with all ASCII values from
280 REM   1 to 255, to print all characters
290 FOR CHAR%=1 TO 255
300   REM Pause in the middle so the characters can be studied
310   IF CHAR%=126 THEN LOCATE 24,8:
       PRINT "PRESS ANY KEY TO SEE MORE";
320   IF CHAR%=126 THEN IF INKEY$="" THEN 320 ELSE CLS
330   CALL SPECPRINT(CHAR%,COLR%)      'Call routine to print
340   REM Space over two columns, drop to left edge &
350   REM   advance two lines if at right margin
360   IF POS(X)+2>40 THEN PRINT:PRINT ELSE LOCATE ,POS(X)+2
370 NEXT CHAR%
380 LOCATE 24,8:PRINT "PRESS ANY KEY TO CONTINUE";
390 A$=INKEY$:IF A$="" THEN 390 ELSE CLS
400 END

```

Lines 130 through 260 must appear before any characters are to be printed. Then the statement **CALL SPECPRINT(char,color)** will print the character with the ASCII value **char** in color **color** at the current cursor location. Char and color must be integers; this is done either by putting percent signs after the variable name, as we have done with **CHAR%** and **COLR%** or by executing an appropriate **DEFINT** statement at the beginning of the program. Char can have any value between 1 and 255 as shown in Appendix B. If char is 67, for example, the character C is printed. Color may be between 0 and 3 in medium-resolution mode or 0 or 1 in high-resolution

mode. In text mode, color must be a valid attribute byte as discussed in Chapter 16 and in the *Technical Reference* manual.

When you execute the statement `CALL SPECPRINT (char,color)`, the last `DEF SEG` statement executed must have been `DEF SEG` with no trailing equal sign or number. If you have executed a different `DEF SEG` since the original `DEF SEG`, which in the program shown in Listing 19-11 occurs on line 130, you must execute `DEF SEG` again.

Note: The cursor location does not change after a character is printed with the machine-language subroutine; if you do not execute a `LOCATE` statement to move the cursor location, the next `PRINT` or `CALL` to the subroutine will erase the previous character.

The program shown in Listing 19-12 demonstrates the use of the color parameter to produce multicolored text in medium-resolution graphics mode. Line 330 selects each of the 15 colors, other than black, for character colors. Color 8 is a gray that may not display clearly unless your screen brightness is high.

ALIGNING THE SCREEN

The left margin of the screen is displayed too far to the left on many television sets and composite monitors. Often the cursor is not even visible when it is in the upper-left corner. This condition can be corrected with the DOS command **MODE,R,T**. This command is typed in response to the DOS `A>` prompt, and causes the display to be shifted to the right. A test pattern is displayed, and you are asked whether the screen is aligned correctly. Type `Y` if it is; if not, type `N`, and the shift is repeated. To shift the display to the left, type:

MODE,L,T and press Enter

in response to the DOS `A>` prompt. The DOS disk must be

Listing 19-12. Print 255 PC Characters in Color.

```

100 REM Program to demonstrate printing all 255
110 REM   PC characters in color.
120 SCREEN 0,1:WIDTH 40:LOCATE ,,0:COLOR 7,0,0:
    CLS:KEY OFF 'Set screen
130 DEF SEG 'Point segment to BASIC interpreter
140 REM Machine-language subroutine will be stored
150 REM   in SUBRT$
160 SUBRT$=""
170 REM Store machine-language subroutine in SUBRT$
180 FOR J=1 TO 26:READ I:SUBRT$=SUBRT$+CHR$(I):NEXT J
190 DATA &h55,&h8B,&hEC,&h8B,&h5E,&h08,&h8A,&h07,&h8B
200 DATA &h5E,&h06,&h8A,&h1F,&hB4,&h09,&h28,&hFF,&hB9
210 DATA &h01,&h00,&hCD,&h10,&h5D,&hCA,&h04,&h00
220 REM Get descriptor address for SUBRT$
230 A=VARPTR(SUBRT$)
240 REM Get string content's address (where machine-
250 REM   language subroutine is) from descriptor
260 ADDR=PEEK(A+1)+PEEK(A+2)*256
270 REM Loop through ASCII values for all 255 characters
280 FOR CHAR%=1 TO 255
290   REM Pause in the middle so characters can be studied
300   IF CHAR%=126 THEN LOCATE 24,8:
       PRINT "PRESS ANY KEY TO SEE MORE";
310   IF CHAR%=126 THEN IF INKEY$="" THEN 310 ELSE CLS
320   REM Select in turn all 16 colors except black
330   COLR%=CHAR% MOD 15 +1
340   CALL ADDR(CHAR%,COLR%) 'Print colored character
350   REM Move two columns to the right, move to left margin
360   REM   and advance two lines if at right margin
370   IF POS(X)+2>40 THEN PRINT:PRINT ELSE LOCATE ,POS(X)+2
380 NEXT CHAR%
390 LOCATE 24,8:PRINT "PRESS ANY KEY TO CONTINUE";
400 A$=INKEY$:IF A$="" THEN 400 ELSE CLS
410 END

```

in drive A. The screen alignment performed with the MODE command remains in effect until the PC is rebooted or turned off.

For more information on the MODE command, or any other DOS command, refer to the *DOS manual*.

VARIABLE SCROLL WINDOW

BASIC usually treats the working screen area as rows 1 through 24 with row 25 reserved for the function keys. It is possible to redefine the top and bottom of the area that

BASIC uses as the working screen for text. To set a new top row for the working area, type:

```
DEF SEG: POKE 91,n and press Enter
```

where **n** is the number of the top line of the working area. Rows above row **n** will not be scrolled.

To set the bottom row of the working area, type:

```
DEF SEG: POKE 92,n and press Enter
```

where **n** is the row number of the bottom of the working area. Only rows between the top and bottom rows will be scrolled. For example, type:

```
CLS: FOR I=65 TO 65+21: PRINT CHR$(I):  
NEXT I and press Enter
```

to put the letters A through V on each row. Now type:

```
DEF SEG: POKE 91,5: POKE 92,15: LOCATE 15
```

and repeatedly press the Enter key to see which portion of the screen scrolls. Anything placed outside the working area, or *scroll window*, remains. Make sure the cursor is in the "working area" before typing additional commands. The LOCATE 15 was executed to put the cursor at the bottom of the scroll window. Values below 1 or above 25, which represent the boundaries of the scroll window, are to be avoided.

When producing graphics, the scroll window concept is most useful. You can establish any section of the screen as an input area which will scroll, and you can produce graphics on the remaining area of the screen that won't scroll. This is similar to our use of row 25 in the *Racecar* game as a stable area of the screen. We can have the best of both worlds: a scrolling input area and a static display area.

The program in Listing 19-13 demonstrates the use of a scroll window with a graphics display. Line 150 sets the top scroll row to row 22, so only the three rows at the bottom of the screen, 22, 23, and 24, are used for text entry and are scrolled. The rest of the screen is used for graphics. This graphics area, which is outside the scroll window,

does not scroll. Note that line 230 restores the normal scroll window at the conclusion of the program. A screen remaining in a three-row scroll window after a program ended would confuse the user. The DEF SEG statement on line 140 must be produced before the scroll window can be modified.

This technique is applicable to any version of Advanced BASIC, but the specific locations 91 and 92 are not guaranteed to work with versions other than 1.10 on the IBM PC.

Listing 19-13. Scroll Window in Medium-Resolution Mode.

```

100 REM Program to demonstrate a scroll window
110 REM   in medium-resolution graphics mode.
120 REM   Text is displayed only on rows 22 thru 24.
130 SCREEN 1,0:COLOR 0,1:CLS:KEY OFF 'Set screen
140 DEF SEG 'Point segment to BASIC interpreter
150 POKE 91,22 'Set top of window to row 23
160 LOCATE 23,1 'Put cursor inside window
170 REM Loop to prompt for figure to draw
180 FOR I=40 TO 280 STEP 40
190   GOSUB 270 'Prompt for figure & draw it
200 NEXT I
210 PRINT:PRINT:PRINT 'Clear the window
220 REM Reset window
230 POKE 91,1
240 LOCATE 24,8:PRINT "PRESS ANY KEY TO CONTINUE";
250 A$=INKEY$:IF A$="" THEN 250 ELSE CLS
260 END
270 REM Subroutine to prompt for figure type
280 REM   and then draw it
290 PRINT:PRINT "PRESS 1 FOR CIRCLE, 2 FOR SQUARE";
300 A$=INKEY$:IF A$="" THEN 300 ELSE A=VAL(A$)
310 REM Try again if not a valid selection
320 IF A>1 AND A<>2 THEN PRINT "INCORRECT-TRY AGAIN":
    GOTO 290
330 REM Draw a circle
340 IF A=1 THEN CIRCLE (I,100),15,2:PAINT STEP(0,0),2:
    PRINT "CIRCLE DRAWN."
350 REM Draw a square
360 IF A=2 THEN LINE(I-15,85)-STEP(30,30),1,BF:
    PRINT "SQUARE DRAWN."
370 RETURN

```


Our thanks to O. B. Canobie; his contribution to the "Star-Dot-Star" column in the Volume 1, Number 5 issue of *PC World* magazine was the basis for this section.

FILL YOUR OWN GRAB BAG

We have presented some useful information about the PC, but there is more waiting to be discovered. If you do much programming, you will eventually need all the knowledge about the PC you can gather. Good sources for further study are: the *Technical Reference* manual, the appendices to the *BASIC* manual, columns such as "Star-Dot-Star" in *PC World*, and user groups. The material in the next chapter will help you understand how some of the methods presented here work.

CHAPTER 20

INSIDE PC GRAPHICS

The commands, programming techniques, and hints described in preceding chapters can all be used without understanding how they work; indeed, one of the strengths of Advanced BASIC is that the user can ignore the machine's hardware and concentrate on programming. However, many people want to know how the PC works and enjoy spending their time exploring the screen, the Basic input/output system (BIOS), DOS, and so on. The knowledge gained often proves to be, not only interesting, but useful as well. For example, the ability to produce color in high-resolution mode, described in the previous chapter, was discovered by such exploration. For those of you ready to learn advanced applications, we will describe the basic workings of graphics on the PC.

Because our area of focus is graphics, we will primarily describe the organization and use of the display screen, as well as several useful points about accessing the entire memory area of the PC. If you are content with the power of Advanced BASIC, you still might want to skim this chapter. An understanding of what is going on inside the computer will aid you in remembering the capabilities and limitations of the various graphics commands which might seem quite arbitrary otherwise.

One more reason to read on: in the "good old days," the microcomputer world consisted of hobbyists and engineers; everyone took pride in "knowing" their computer "inside and out." Today, many of us feel it is most exciting and rewarding to work with the computer at the

machine level, even if that opinion does relegate us to the “computer nerd” or “bytehead” categories!

Before we begin, we should mention that IBM’s *Technical Reference* manual is an invaluable guide to the inner workings of the PC.

SOME USEFUL TERMS

We will use the term *byte* throughout this chapter. A byte is simply one memory location in the PC. A byte consists of eight binary digits (digits in base 2), called *bits*. Each bit can have the value 0 or 1, and each byte can have any integer (whole number) value in the range 0 to 255. You may recall that there are 255 characters available on the PC; this is no coincidence, but occurs because one byte holds one character, and the value of a byte can be no higher than 255.

Each memory location is referenced by an *address*. An address is a number which specifies a single memory location. Addresses might be thought of as somewhat analogous to array subscripts, in that address 0 selects the first location, 1 the second location, and so on. It is customary to specify addresses in hexadecimal (base 16) notation, with the letters A through F used as digits in addition to 0 through 9 used in decimal (base 10) notation. In general, we will follow convention and give addresses in hexadecimal notation, so don’t be surprised to see letters mixed in with numbers. It’s not necessary for you to know anything about hexadecimal notation to benefit from this chapter, but you should acquire at least a passing familiarity with it if you want to do advanced work with microcomputers.

In BASIC, hexadecimal numbers are prefixed with **&H**. For example, the hexadecimal number B800 is stored in the variable **POINTER** with the command **POINTER = &HB800**

The letter K, following a number, is usually used to mean “about 1000.” 1K is equal to 1024. This is because computers use binary (base 2) notation, so numbers are often powers of 2, such as 8, 16, 64, 256, 1024, and so on. Incidentally, the reason hexadecimal notation is often

used in the computer world is that it is easy to translate from hexadecimal (which people can use) to binary (which computers use) and back, while it is not easy to do so with decimal notation. Important areas of memory tend to begin at addresses that are powers of 2, while addresses, device control bytes, and screen memory bytes are most easily expressed in binary or hexadecimal notation.

MEMORY-MAPPED VIDEO

The IBM PC's display screen is linked to an area of the computer's memory. The Color/Graphics Adapter constantly scans this area of memory to generate the video signal that makes the image appear on the screen. The contents of this memory form the basis for whatever appears on the screen, with each memory location, or byte, corresponding to a screen location. In text mode, for example, a value of 65 in the first byte of the Color/Graphics Adapter screen memory would cause the character A to appear in the upper-left corner of the screen. Thus, the PC's display is called a *memory-mapped* display, because the contents of an area of memory "map out" what is to be shown on the screen.

The concept of the memory map explains how BSAVE and BLOAD work. BSAVE stores the contents of a specified area of memory to disk, while BLOAD restores the information from disk to an area of memory. Because the screen memory map is an area of memory, BSAVE and BLOAD can manipulate the screen information as readily as the contents of any other memory location. We BSAVE the contents of the screen memory area to disk, then BLOAD the saved information back into the screen memory to restore the display.

There are separate memory-map locations for the monochrome display and Color/Graphics Adapter screens. The monochrome screen begins at address B000:0000; the Color/Graphics Adapter screen begins at address B800:0000. (Remember that we are using hexadecimal notation.)

Two numbers separated by a colon is the standard way to specify addresses with the 8088 microprocessor chip. In

order to address the PC's potential one megabyte of memory (that is, to specify a memory location from among the approximately one million available on the PC), the 8088 uses two values. One is a *base* value, which defines a segment, and the other is an *offset* from that base value. The offset value can address a maximum of 64K, or about 65,000 bytes of memory relative to the segment value (which can point anywhere in the one-megabyte memory space of the PC). This is why BASIC can use only a 64K work space—a segment register is set when the program is started and is left unchanged so only the 64K addressable memory via the offset value is available. An address of the form B800:0000 means the location pointed to by segment value B800 and offset value 0000. We will discuss shortly to what memory location a segment:offset pair points.

ACCESSING MEMORY FROM BASIC

BASIC allows the programmer to specify both segment and offset values to get data into and out of the specified location. Thus, the programmer can access any location in memory, including, as described previously, the screen memory areas.

The statement **DEF SEG=n** sets the segment value, where the optional parameter **n** is a value between 0 and 65535 or between &H0000 and &HFFFF. (The two ranges are equivalent.) The space between DEF and SEG is required as is the equal sign if **n** is specified. For example, **DEF SEG=&HB800** sets the segment value to point to the beginning of the Color/Graphics Adapter screen memory map. Remember that this is the first statement we used when saving and restoring the screen. This DEF SEG= is necessary so that subsequent commands will operate in the screen memory area.

DEF SEG, with no value or equal sign, is a valid statement and sets the segment to point to the BASIC program itself. Thus, when we set the scroll window values in Chapter 19, we modified the version of Advanced BASIC that was in memory. Because the BASICA.COM program on the disk remained unchanged, the modification was only temporary.

The segment value set with the DEF SEG statement remains in effect until another DEF SEG is executed or until BASIC is exited with the SYSTEM command, so you do not have to execute a DEF SEG for every BSAVE or BLOAD. However, extra DEF SEG statements do no harm. If you frequently change the segment value, it's a good practice to be sure your programs are pointing to the correct area of memory.

The PEEK and POKE statements access a specific memory location relative to the segment defined with DEF SEG. The statement **PEEK(address)** returns the value stored at location *address*, where *address* is an offset value relative to the defined segment, in the range 0 to 65535. For example, **DEF SEG=&HB800: PRINT PEEK(8)** prints the value of the byte at address B800:0008, the ninth location in the Color/Graphics Adapter memory map. (Remember that the first location is location B800:0000.)

The statement **POKE address,i** puts the value *i* into the memory locations *address*, where *address* is relative to the current segment as assigned by the last DEF SEG statement. For example, select text mode and type:

```
DEF SEG=&B800: POKE 0,66 and press Enter
```

to put the value 66 into the first location in the Color/Graphics Adapter memory. This location corresponds to the upper-left corner of the screen, and, indeed, the letter B appears in that location on your screen.

The manner in which a segment:offset pair specifies a memory address is not as simple as it might be. There are 1024K potential memory locations in the PC, with the actual addresses ranging from 0 to FFFFFF (hexadecimal). The memory location addressed by a segment:offset address is determined by multiplying the segment value by 16 and adding that product to the offset. This process is equivalent to putting an additional zero (hexadecimal) at the right end of the segment value and then adding the resulting value to the offset. For example, 0100:0050 points to 01000 + 0050 = address 1050 (all in hexadecimal). 0000:1050 and 0105:0000 would also point to

address 1050 (hexadecimal). Obviously, many different combinations of segment:offset can point to the same actual memory location.

PEEKing around the PC with various segment values is a great way to explore the workings of the machine. For example, the interrupt vectors, which indicate the locations of input, output, and disk functions, start at 0000:0000. The BIOS (Basic Input/Output System) variables start at 0040:0000. These memory locations select which the text goes to, store Caps Lock and Num Lock states, disk parameters, cursor location, the key buffer, and much more. When we toggled the state of the Num Lock and Caps Lock keys in Chapter 19, we modified the memory location in which the PC stores the states of these keys.

DOS, which provides high-level support to programs, begins at 0060:0000. The BIOS itself, with all BASIC input/output functions, starts at F000:E000, while Cassette BASIC, much of which is used by Advanced BASIC, starts at F000:6000.

DOS, Cassette BASIC, and the BIOS can all be explored with the DEBUG.COM program, but you will need to learn a little about Assembly language before you start exploring. The *DOS* manual is a good reference for DOS, while IBM's *Technical Reference* manual lists the Assembly language source program for the BIOS, along with other useful information regarding the internal workings of the PC. There is, unfortunately, no good reference for the internal workings of Cassette BASIC.

Although all of the above is useful for the most ambitious, those of us concerned primarily with exploring the graphics screens from BASIC need only remember that **DEF SEG=&HB800** sets the segment pointer to the beginning of the Color/Graphics Adapter memory map. **DEF SEG=&HB000** sets the segment pointer to the beginning of the Monochrome Adapter memory map.

SCREEN MEMORY ORGANIZATION—TEXT MODE

Now that we know where the screen memory is, we need to discuss just how that memory translates onto the display screen.

Each memory location is one byte in size. What this byte causes to be displayed on the screen depends on the mode the screen is in. In text mode, each even-numbered byte in the screen memory map defines a character to be displayed. A value in an even-numbered byte causes the corresponding ASCII character to be displayed. (See Appendix B for the full set of characters and ASCII values.) Each odd byte contains the attribute for the character, controlling the foreground, background, and blink characteristics of the character, as discussed in Chapter 16. There are 1000 characters on the screen in 40-column mode, and, because each character requires two bytes, 2000 bytes are needed for a 40-column display in text mode. The bytes at offsets 0 and 1 control the character in the upper-left corner. (When we are discussing offsets in the screen memory map, we will use decimal numbers.) The bytes at 2 and 3 control the character at row 1, column 2. This continues for 40 columns to the right edge of the screen, then the next bytes control the character at row 2, column 1, and so on, one row at a time to the bottom of the screen. For example, in 40-column color text mode type:

```
DEF SEG=&HB800: POKE 0,65: POKE 2,66: POKE 4,67
and press Enter
```

and the characters ABC instantly appears, left to right on the top row. Type:

```
POKE 80,89 and press Enter
```

and Y appears on the second row. Type:

```
POKE 1998,90 and press Enter
```

and Z appears at the lower-right corner. This is the last location in the 40-column text-mode screen memory map.

The attribute byte can also be changed. For example, type:

```
POKE 1,1 and press Enter
```

sets character A at row 1, column 1 to the color blue. Simi-

larly, **POKE 5,32** causes the C at row 1, column 3 to be displayed in black against a green background.

The 80-column mode is similar to 40-column mode. The memory map is 4000 bytes, rather than 2000 bytes, and 80 pairs of bytes represent each row. For example, select 80-column mode and type:

```
DEF SEG & HB800:POKE 0,65: POKE 160,66: POKE
3998,67 and press Enter
```

The character C is at the last character location in the 80-column text-mode screen memory map.

You can read values back out of the screen memory as easily as you put them in. Select 40-column mode, then type:

POKE 10,65: PRINT CHR\$(PEEK(10)) puts the character A on the screen, reads its ASCII value from memory, converts it to a character, and PRINTs it.

For reasons to be explained subsequently, there are over 16,000 bytes of memory on the Color/Graphics Adapter. Yet, text mode requires as few as 2000 and no more than 4000 bytes of memory. The extra memory is used to provide the extra screens that are available in text mode. For example, in 40-column text mode, the first 2000 bytes are used for screen 0. 48 bytes are not used, then the next 2000 bytes, starting at offset 2048, are used for screen 1. In 40-column text mode, clear the screen and type:

```
DEF SEG=&HB800: POKE 2048,ASC('*') and press Enter
```

(The ASC(x) function converts the character x to its ASCII value, so that ASC('A') would, for example, return 65.) Nothing appears. Now type:

```
SCREEN,,1,1 and press Enter
```

to switch to screen page 1, and there is our missing asterisk! There are 6 other 2000-byte memory maps starting at offsets 4096, 6144, 8192, 10240, 12288, and 14336, respectively.

In 80-column text mode, there are 4 memory maps starting at offsets 0, 4096, 8192, and 12288, respectively. For example, type:

```
SCREEN,,0,0 and press Enter
```

to return to page 0, and type:

`WIDTH 80` and press Enter

to select 80-column mode. (If you have exited BASIC since the last DEF SEG, you will have to type DEF SEG=&HB800 again.) Now type:

`POKE 8192,ASC("@")` and press Enter

Now type:

`SCREEN,,2,2` and press Enter

and there's our @ on screen 2. Type:

`SCREEN,,0,0` and press Enter

to return to screen page 0.

SCREEN MEMORY ORGANIZATION—HIGH-RESOLUTION MODE

In high-resolution graphics mode there are 128,000 (640 x 200) pixels on the screen. Each pixel may be 0 for black or 1 for white. One bit can have the value 0 or 1, so one bit can represent one pixel in high-resolution mode. There are 8 bits in a byte, so one byte can control 8 pixels. Therefore, the high-resolution screen memory map requires 16,000 (128,000 pixels divided by 8 pixels per byte) bytes of memory. This is why there are 16K bytes of memory on the Color/Graphics Adapter.

The 8 pixels controlled by any byte are horizontally adjacent to one another. Bit 7 (the leftmost bit) controls the leftmost pixel, bit 6 controls the next pixel, and so on. If a bit value is 0, then the corresponding pixel is black; if the value is 1, then the pixel is white. The first byte in the high-resolution memory map, the byte at offset 0 or byte number 0, controls the 8 leftmost pixels in the top row (columns 0-7), as shown in Fig. 20-1. Byte 1 controls the next 8 pixels to the right (columns 8-15), and so on, for 80 bytes. Byte number 79 controls the 8 pixels at columns 632-639 of the top row (row 0).

UPPER-LEFT CORNER OF SCREEN

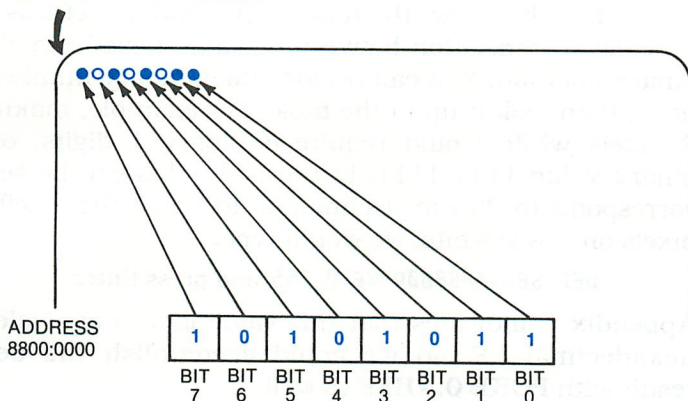


Fig. 20-1. Pixels controlled by the first byte in graphics-screen memory map—high resolution.

For example, select high-resolution mode with **SCREEN 2** and type:

DEF SEG=&HB800 and press Enter

Now, whatever you **POKE** into memory in the address ranges 0 through 7999 and 8192 through 16191 immediately affects what is displayed on the high-resolution screen. (We will explain why there are two ranges shortly.) Type:

POKE 0,1 and press Enter

and the dot at column 7 of row 0 becomes white. Type:

POKE 1,1 and press Enter

and the dot at column 15 is white. **POKE 2,255** turns on all 8 pixels, at columns 16-23, controlled by the third byte. **POKE 79,170** turns on every other pixel controlled by byte 79.

Familiarity with binary notation is necessary to work effectively with the graphics screen. Hexadecimal notation is equally essential. We suggest you familiarize yourself with these number systems and experiment with **PEEK** and **POKE** in the graphics mode. You will find that working on practical projects is the quickest way to acquire

proficiency. For now, the table in Appendix C of this book provides a translation between decimal, hexadecimal, and binary notation. You can decide which binary number you need, then look it up in the table. For example, making all 8 pixels white would require 8 binary 1 digits, or the binary value 1111 1111. In the table, this can be seen to correspond to 255 in decimal, so to make the 8 leftmost pixels on row 0 white, we would type:

DEF SEG=&HB800POKE 0,255 and press Enter

Appendix C indicates that this value is also equivalent to hexadecimal FF, so we could accomplish the desired result with **POKE 0,&HFF** as well.

There is a slight complication at this point. One might logically think that the byte at offset 80 would control the first 8 pixels on row one, but this is not the case. Type:

POKE 80,255 and press Enter

and you will see that byte 80 controls the first 8 pixels on row two. The byte at offset 160 controls the first 8 pixels on row four, and so on down to the byte at offset 7920, which controls the first 8 pixels on row 98. Thus, all the pixels on even rows are controlled by the bytes in the offset range of 0 through 7999 on the Color/Graphics Adapter.

The pixels on odd rows are controlled by the bytes in the address range 8192 (B800:2000 hexadecimal) to 16191. The eight pixels on row 1 are controlled by the bytes in the address range 8192 (B800:2000 hexadecimal) to 16191. The eight pixels on row 1 (the second row down), columns 0-7, are controlled by the byte at offset 8192. The next eight pixels (columns 8-15) are controlled by the byte at offset 8193. In essence, the odd rows are addressed like the even row just above them except that 8192 must be added to the offset.

For example, make sure the segment is set to B800 and type:

POKE 8192,192 and press Enter

to turn on the leftmost pixel on row 1. **POKE 8192+79,85** turns on every other pixel at the right end of row 1.

The last pixels on an even row are controlled by the byte at offset 7999. Type:

```
POKE 7999,255 and press Enter
```

to make these pixels white. The last pixels on the last odd row are controlled by the byte at offset 16191 (8192 + 7999). Type:

```
POKE 16191,255 and press Enter
```

to make these pixels white. Note that **POKE 16191,0** will make these pixels black again.

Clear the screen and type:

```
FOR I=0 TO 320: POKE I,51: NEXT and press Enter
```

to draw several dotted lines. Notice that only every other line is drawn because we are operating only within the even-line section of the high-resolution memory map. Type:

```
FOR I=0 TO 320: POKE 8192+I,204: NEXT and press Enter
```

to draw several dotted lines on odd rows. The dots on the odd rows are offset from the dots on the even rows because the value 204 produces an 8-pixel pattern that is exactly the opposite of that produced by 51.

We can see the odd-even line separation with the BLOAD statement. Type:

```
LINE (220,50)-(420,150),,BF and press Enter
```

to draw a solid box, then save it with:

```
DEF SEG=&HB8000: BSAVE "BOX",0,16384 and press Enter
```

(Note that we save 16,384 bytes because this is the number of bytes in the high-resolution screen. The 16,384 is equal to &H4000, which we used in the previous chapter for the number of bytes to save in the graphics screen.) Now type:

```
CLS: BLOAD "BOX" and press Enter
```

The even lines appear first, and, after a moment, the odd

lines appear as well. BLOAD transfers a disk file to memory, and floppy disks are slow enough so that we can see the transfer in progress. The transfer proceeds by replacing the byte at offset 0, then offset 1, and so on, up to offset 16383. Because all the even lines are in the offset range 0-7999, they are replaced before any of the odd lines in the range 8192-16191.

You can test the state of a pixel by PEEKing the appropriate byte. For example, **POKE 0,1: PRINT PEEK(0)** turns on the pixel at row 0, column 7, and then returns the value 1 for the 8 pixels controlled by the byte at offset 0. The value 1 indicates that only the rightmost pixel of the 8 is white.

SCREEN MEMORY ORGANIZATION—MEDIUM-RESOLUTION MODE

In medium-resolution graphics mode there are 64,000 (320 x 200) pixels on the screen. Each pixel may be 0 for the background color or 1 through 3 for the colors in the selected palette. Two bits are required to represent the values 0 through 3, as shown in Appendix C, which means that four pixels can be controlled by a single byte. Therefore, the medium-resolution screen memory map requires 16,000 (64,000 pixels divided by 4 pixels per byte) bytes of memory.

The 4 pixels controlled by any byte are horizontally adjacent to one another. Bit 7 (the leftmost bit) and bit 6 control the leftmost pixel, bits 5 and 4 control the next pixel, and so on. If the bit pair value is 0, the pixel is the background color; if the value is not zero, the pixel is the corresponding color from the current palette. The first byte in the medium-resolution memory map, the byte at offset 0 or byte number 0, controls the 4 leftmost pixels on the top row (columns 0-3), as shown in Fig. 20-2. Byte 1 controls the next 4 pixels to the right (columns 4-7), and so on, for 80 bytes. The byte at offset 79 controls the 4 pixels at columns 316-319 of the top row (row 0).

For example, type:

```
SCREEN 1,0: COLOR 0,1: CLS and press Enter
```


to select medium-resolution color-graphics mode with palette 1. Then type:

DEF SEG=&HB8000: POKE 0,3 and press Enter

to draw a white pixel in row 0, column 3. This is the rightmost pixel controlled by the byte at offset 0. **POKE 1,2** draws a magenta pixel at row 0, column 2. (If this does not show well on your display, type:

POKE 1,10 and press Enter

to draw two adjacent magenta pixels.) **POKE 2,1** draws a cyan dot. (Alternatively, **POKE 2,5** draws a doubled cyan dot.) Finally, **POKE 79,255** makes all four pixels controlled by the byte at offset 79 (row 0, columns 316-319) white.

UPPER-LEFT CORNER OF SCREEN

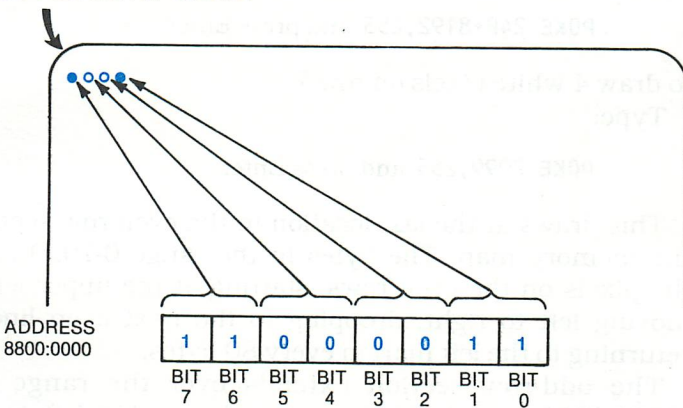


Fig. 20-2. Pixels controlled by the first byte in graphics screen memory map—medium resolution.

Remember that you can use Appendix C of this book to translate your binary numbers into more easily usable decimal numbers. For instance, four blue pixels would mean four pixel pairs with the value 1, or 01 01 01 01 which is 0101 0101. Appendix C shows this to be equal to 85 in decimal notation, so to make the first 4 pixels on row 0 cyan, we would type:

POKE 0,85 and press Enter

In medium-resolution mode, the odd and even rows are kept in separate areas of memory, as they are in high-resolution mode. The byte at offset 80 controls the leftmost 4 pixels on row 2, the byte at offset 160 controls the first 4 pixels on row 4, and the byte at offset 7920 controls the first 4 pixels on row 198. The pixels in an odd row are 8192 bytes further into the memory map than the pixels on the even row above, as they are in high-resolution mode. The first byte in the odd-lines section of the screen memory map controls row 1, columns 0-3 and is located at offset 8192. The next 4 pixels to the right are controlled by the byte at offset 8193. For example, type:

```
POKE 240,255 and press Enter
```

to draw 4 white pixels on row 6 and then type:

```
POKE 240+8192,255 and press Enter
```

to draw 4 white pixels on row 7.

Type:

```
POKE 7999,255 and press Enter
```

This draws at the last location in the even-row section of the memory map. The bytes in the range 0-7999 control the pixels on the even rows, starting at the upper left and moving left to right, dropping to the next even line and returning to the left margin every 80 bytes.

The odd-row section extends over the range 8192 (B800:2000 hexadecimal) to 16191, moving left to right and down as with the even lines. Type:

```
POKE 16191,255 and press Enter
```

to draw white pixels at the last location of the odd rows memory map.

Type:

```
FOR I=800 TO 1600: POKE I,10: NEXT and press Enter
```

to draw several dotted magenta lines on even rows. Note that the lines wrap from the right margin to the left end of

the next even row down. Similarly:

```
FOR I=800+8192 TO 1600+8192: POKE I,5: NEXT and  
press Enter
```

draws several dotted cyan lines on odd rows.

The PEEK statement can be used to determine the state of pixels in the medium-resolution screen. For example, type:

```
POKE 0,5: PRINT PEEK(0) and press Enter
```

The 5 printed on the screen indicates that the pixels at row 0, columns 2 and 3 are cyan, and columns 0 and 1 are black.

WHY PEEK AND POKE?

You'll probably never need PEEK and POKE to do graphics from BASIC, because PSET, POINT, and the more advanced graphics commands do everything you might need. Most languages, however, lack graphics commands; to produce graphics from these languages you will have to construct your own set of subroutines to perform the necessary functions using the equivalent of PEEK and POKE. This is done, for example, when writing arcade-style games in Assembly language.

To see why Assembly language is used, draw a line using POKE and a FOR . . . NEXT loop, then draw the same line with the LINE statement. The difference in speed is the difference between the speed of execution in BASIC and in Assembly language—in general, Assembly language is many times faster. When writing your own graphics routines in Assembly language, you must account for such characteristics as the separation of odd and even lines in the screen memory map; all the information (and more) discussed here is needed. The programming is not easy, but, on the other hand, complete command of BASIC, Assembly language, and the internal workings of the computer allows you to get the most from your PC. Five years ago, it was considered unlikely that arcade-style games or language compilers such as FOR-

TRAN could ever work on a microcomputer; today, there are hundreds of each. See how far you can stretch the possibilities of your PC!

One final note regarding advanced techniques: the BIOS (Basic Input/Output System)—which is permanently stored in ROM and always available to the programmer—provides a wide range of disk, keyboard, video, and other functions. Included in the video section is the ability to LOCATE (or place) the cursor, determine the cursor location, switch the screen mode, scroll the screen up or down, put a character with any attribute on the screen, select the active screen page, select the palette and the background and border colors, plot a dot at an x,y coordinate, read the state of the dot at an x,y coordinate, and read a light pen position.

This book is not intended for the most advanced programmer, and the use of the BIOS is an advanced technique. The BIOS can be accessed via a machine-language subroutine from BASIC, FORTRAN, or other high-level language, or can be accessed directly via an interrupt from Assembly language. If you are looking for a new direction in which to explore, learning to access the BIOS from BASIC, or learning Assembly language along with the BIOS could be a rewarding experience. Appendix A to IBM's *Technical Reference* manual will help in this pursuit. Appendix C of the *BASIC* manual provides some information as well, although it is by no means a complete discussion of machine-language subroutines. *The 8086 Book*, by Russell Rector and George Alexy, and *The 8086 Primer*, by Stephen Morse, are good references on Assembly language for the PC. IBM's Macro Assembler and the accompanying manual are necessary to do any work with Assembly language. Numerous magazine articles address the use of machine-language subroutines from BASIC, Assembly language, and the BIOS.

If you master Assembly language and the BIOS, congratulations! You will be an expert on the PC, well-equipped for any task. The only way to get there is to read and experiment, so—go forth!

APPENDIX A

THE SET OF CHARACTERS AVAILABLE FROM BASIC

Following is a list of the 256 characters that can be produced with the **CHR\$** function and are known as the ASCII character set for the PC. Note that some of the values do not produce a visible character, but rather have an effect such as moving the cursor.

Graphics for the IBM PC

ASCII value	Character	Control character	ASCII value	Character
000	(null)	NUL	032	(space)
001	☺	SOH	033	!
002	☻	STX	034	"
003	♥	ETX	035	#
004	♦	EOT	036	\$
005	♣	ENQ	037	%
006	♠	ACK	038	&
007	(beep)	BEL	039	'
008	■	BS	040	(
009	(tab)	HT	041)
010	(line feed)	LF	042	*
011	(home)	VT	043	+
012	(form feed)	FF	044	,
013	(carriage return)	CR	045	-
014	🎵	SO	046	.
015	☼	SI	047	/
016	▶	DLE	048	0
017	◀	DC1	049	1
018	↕	DC2	050	2
019	!!	DC3	051	3
020	☐	DC4	052	4
021	§	NAK	053	5
022	▬	SYN	054	6
023	↕	ETB	055	7
024	↑	CAN	056	8
025	↓	EM	057	9
026	→	SUB	058	:
027	←	ESC	059	;
028	(cursor right)	FS	060	<
029	(cursor left)	GS	061	=
030	(cursor up)	RS	062	>
031	(cursor down)	US	063	?

ASCII value	Character
064	@
065	A
066	B
067	C
068	D
069	E
070	F
071	G
072	H
073	I
074	J
075	K
076	L
077	M
078	N
079	O
080	P
081	Q
082	R
083	S
084	T
085	U
086	V
087	W
088	X
089	Y
090	Z
091	[
092	\
093]
094	^
095	_

ASCII value	Character
096	`
097	a
098	b
099	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	
125	}
126	~
127	␣

ASCII value	Character	ASCII value	Character
128	Ç	160	á
129	ü	161	í
130	é	162	ó
131	â	163	ú
132	ä	164	ñ
133	à	165	Ñ
134	å	166	<u>a</u>
135	ç	167	<u>o</u>
136	ê	168	¿
137	ë	169	┌
138	è	170	└
139	ï	171	½
140	î	172	¼
141	ì	173	¡
142	Ä	174	«
143	Å	175	»
144	É	176	░░░░
145	æ	177	▤▤▤▤
146	Æ	178	▩▩▩▩
147	ô	179	
148	ö	180	├
149	ò	181	┼
150	û	182	┤
151	ù	183	├
152	ÿ	184	┼
153	Ö	185	┼
154	Ü	186	
155	¢	187	┼
156	£	188	┼
157	¥	189	┼
158	Pt	190	┼
159	f	191	└

ASCII value	Character	ASCII value	Character
192	Ł	224	α
193	ł	225	β
194	Ť	226	Γ
195	ť	227	π
196	—	228	Σ
197	+	229	σ
198	ƒ	230	μ
199	ƒ	231	τ
200	ℓ	232	Φ
201	ℓ	233	⊖
202	≡	234	Ω
203	≡	235	δ
204	≡	236	∞
205	≡	237	∅
206	≡	238	€
207	≡	239	∩
208	≡	240	≡
209	≡	241	±
210	≡	242	≥
211	ℓ	243	≤
212	ℓ	244	ƒ
213	ℓ	245	J
214	ℓ	246	÷
215	≡	247	≈
216	≡	248	°
217	┘	249	•
218	┘	250	•
219	■	251	√
220	■	252	n
221	■	253	²
222	■	254	■
223	■	255	(blank 'FF')

APPENDIX B

THE FULL 255 CHARACTER SET OF THE IBM PC

Following is a list of the 255 characters the IBM PC can display. These characters are the ASCII character set of the PC. Not all of these characters can be displayed directly from DOS or BASIC since, in those uses, they perform special functions such as beeping or moving the cursor. All are available through the BIOS function INT 10 (hexadecimal) from Assembly or machine language. See the *Technical Reference* manual for more information on the PC character set.

Reprinted by permission from *Technical Reference* © 1981 by International Business Machines Corporation.

DECIMAL VALUE	➡	0	16	32	48	64	80	96	112
⬇	HEXA- DECIMAL VALUE	0	1	2	3	4	5	6	7
0	0	BLANK (NULL)	▶	BLANK (SPACE)	0	@	P	'	p
1	1	😊	◀	!	1	A	Q	a	q
2	2	😄	↕	"	2	B	R	b	r
3	3	♥	!!	#	3	C	S	c	s
4	4	♦	¶	\$	4	D	T	d	t
5	5	♣	§	%	5	E	U	e	u
6	6	♠	▬	&	6	F	V	f	v
7	7	•	↕	'	7	G	W	g	w
8	8	●	↑	(8	H	X	h	x
9	9	○	↓)	9	I	Y	i	y
10	A	◯	→	*	:	J	Z	j	x
11	B	♂	←	+	;	K	l	k	{
12	C	♀	└	,	<	L	\	l	!
13	D	🎵	↔	—	=	M	l	m	}
14	E	🎵	▲	.	>	N	^	n	~
15	F	☀	▼	/	?	O	—	o	△

Graphics for the IBM PC

DECIMAL VALUE	➡	128	144	160	176	192	208	224	240
➡	HEXA- DECIMAL VALUE	8	9	A	B	D	C	E	F
0	0	℥	Ē	á	1/4 Dots On			∞	≡
1	1	ü	Æ	í	1/2 Dots On			β	±
2	2	é	FE	ó	3/4 Dots On			γ	≥
3	3	â	ô	ú				π	≤
4	4	ä	ö	ñ				Σ	∫
5	5	à	ò	Ñ				σ	
6	6	å	û	ä				μ	÷
7	7	ç	ù	ó				τ	≈
8	8	ê	ÿ	¿				Φ	°
9	9	ë	Ö	┐				⊖	•
10	A	è	Û	└				Ω	•
11	B	ï	¢	1/2				δ	√
12	C	î	£	1/4				∞	η
13	D	ì	¥	ì				∅	²
14	E	Ä	Pts	«				∈	■
15	F	Å	f	»				∩	BLANK 'FF'

APPENDIX C

DECIMAL, HEXADECIMAL, AND BINARY CONVERSION TABLE

This table shows the values 0 through 255 in binary (base 2), hexadecimal (base 16), and decimal (base 10) notation. Read across horizontally to translate from one notation to another.

Decimal	Hex	Binary	Decimal	Hex	Binary
0	0	0000 0000	25	19	0001 1001
1	1	0000 0001	26	1A	0001 1010
2	2	0000 0010	27	1B	0001 1011
3	3	0000 0011	28	1C	0001 1100
4	4	0000 0100	29	1D	0001 1101
5	5	0000 0101	30	1E	0001 1110
6	6	0000 0110	31	1F	0001 1111
7	7	0000 0111	32	20	0010 0000
8	8	0000 1000	33	21	0010 0001
9	9	0000 1001	34	22	0010 0010
10	A	0000 1010	35	23	0010 0011
11	B	0000 1011	36	24	0010 0100
12	C	0000 1100	37	25	0010 0101
13	D	0000 1101	38	26	0010 0110
14	E	0000 1110	39	27	0010 0111
15	F	0000 1111	40	28	0010 1000
16	10	0001 0000	41	29	0010 1001
17	11	0001 0001	42	2A	0010 1010
18	12	0001 0010	43	2B	0010 1011
19	13	0001 0011	44	2C	0010 1100
20	14	0001 0100	45	2D	0010 1101
21	15	0001 0101	46	2E	0010 1110
22	16	0001 0110	47	2F	0010 1111
23	17	0001 0111	48	30	0011 0000
24	18	0001 1000	49	31	0011 0001

Graphics for the IBM PC

Decimal	Hex	Binary	Decimal	Hex	Binary
50	32	0011 0010	96	60	0110 0000
51	33	0011 0011	97	61	0110 0001
52	34	0011 0100	98	62	0110 0010
53	35	0011 0101	99	63	0110 0011
54	36	0011 0110	100	64	0110 0100
55	37	0011 0111	101	65	0110 0101
56	38	0011 1000	102	66	0110 0110
57	39	0011 1001	103	67	0110 0111
58	3A	0011 1010	104	68	0110 1000
59	3B	0011 1011	105	69	0110 1001
60	3C	0011 1100	106	6A	0110 1010
61	3D	0011 1101	107	6B	0110 1011
62	3E	0011 1110	108	6C	0110 1100
63	3F	0011 1111	109	6D	0110 1101
64	40	0100 0000	110	6E	0110 1110
65	41	0100 0001	111	6F	0110 1111
66	42	0100 0010	112	70	0111 0000
67	43	0100 0011	113	71	0111 0001
68	44	0100 0100	114	72	0111 0010
69	45	0100 0101	115	73	0111 0011
70	46	0100 0110	116	74	0111 0100
71	47	0100 0111	117	75	0111 0101
72	48	0100 1000	118	76	0111 0110
73	49	0100 1001	119	77	0111 0111
74	4A	0100 1010	120	78	0111 1000
75	4B	0100 1011	121	79	0111 1001
76	4C	0100 1100	122	7A	0111 1010
77	4D	0100 1101	123	7B	0111 1011
78	4E	0100 1110	124	7C	0111 1100
79	4F	0100 1111	125	7D	0111 1101
80	50	0101 0000	126	7E	0111 1110
81	51	0101 0001	127	7F	0111 1111
82	52	0101 0010	128	80	1000 0000
83	53	0101 0011	129	81	1000 0001
84	54	0101 0100	130	82	1000 0010
85	55	0101 0101	131	83	1000 0011
86	56	0101 0110	132	84	1000 0100
87	57	0101 0111	133	85	1000 0101
88	58	0101 1000	134	86	1000 0110
89	59	0101 1001	135	87	1000 0111
90	5A	0101 1010	136	88	1000 1000
91	5B	0101 1011	137	89	1000 1001
92	5C	0101 1100	138	8A	1000 1010
93	5D	0101 1101	139	8B	1000 1011
94	5E	0101 1110	140	8C	1000 1100
95	5F	0101 1111	141	8D	1000 1101

Decimal	Hex	Binary	Decimal	Hex	Binary
142	8E	1000 1110	188	BC	1011 1100
143	8F	1000 1111	189	BD	1011 1101
144	90	1001 0000	190	BE	1011 1110
145	91	1001 0001	191	BF	1011 1111
146	92	1001 0010	192	C0	1100 0000
147	93	1001 0011	193	C1	1100 0001
148	94	1001 0100	194	C2	1100 0010
149	95	1001 0101	195	C3	1100 0011
150	96	1001 0110	196	C4	1100 0100
151	97	1001 0111	197	C5	1100 0101
152	98	1001 1000	198	C6	1100 0110
153	99	1001 1001	199	C7	1100 0111
154	9A	1001 1010	200	C8	1100 1000
155	9B	1001 1011	201	C9	1100 1001
156	9C	1001 1100	202	CA	1100 1010
157	9D	1001 1101	203	CB	1100 1011
158	9E	1001 1110	204	CC	1100 1100
159	9F	1001 1111	205	CD	1100 1101
160	A0	1010 0000	206	CE	1100 1110
161	A1	1010 0001	207	CF	1100 1111
162	A2	1010 0010	208	D0	1101 0000
163	A3	1010 0011	209	D1	1101 0001
164	A4	1010 0100	210	D2	1101 0010
165	A5	1010 0101	211	D3	1101 0011
166	A6	1010 0110	212	D4	1101 0100
167	A7	1010 0111	213	D5	1101 0101
168	A8	1010 1000	214	D6	1101 0110
169	A9	1010 1001	215	D7	1101 0111
170	AA	1010 1010	216	D8	1101 1000
171	AB	1010 1011	217	D9	1101 1001
172	AC	1010 1100	218	DA	1101 1010
173	AD	1010 1101	219	DB	1101 1011
174	AE	1010 1110	220	DC	1101 1100
175	AF	1010 1111	221	DD	1101 1101
176	B0	1011 0000	222	DE	1101 1110
177	B1	1011 0001	223	DF	1101 1111
178	B2	1011 0010	224	E0	1110 0000
179	B3	1011 0011	225	E1	1110 0001
180	B4	1011 0100	226	E2	1110 0010
181	B5	1011 0101	227	E3	1110 0011
182	B6	1011 0110	228	E4	1110 0100
183	B7	1011 0111	229	E5	1110 0101
184	B8	1011 1000	230	E6	1110 0110
185	B9	1011 1001	231	E7	1110 0111
186	BA	1011 1010	232	E8	1110 1000
187	BB	1011 1011	233	E9	1110 1001

Graphics for the IBM PC

Decimal	Hex	Binary	Decimal	Hex	Binary
234	EA	1110 1010	245	F5	1111 0101
235	EB	1110 1011	246	F6	1111 0110
236	EC	1110 1100	247	F7	1111 0111
237	ED	1110 1101	248	F8	1111 1000
238	EE	1110 1110	249	F9	1111 1001
239	EF	1110 1111	250	FA	1111 1010
240	F0	1111 0000	251	FB	1111 1011
241	F1	1111 0001	252	FC	1111 1100
242	F2	1111 0010	253	FD	1111 1101
243	F3	1111 0011	254	FE	1111 1110
244	F4	1111 0100	255	FF	1111 1111

GLOSSARY

Address. Similar to the street address for a house; each location in memory is referred to by a unique number, known as its address.

Analog. Circuitry that operates with electrical signals that can represent a continuous range of values. See digital.

Array. String of variables referred to with the same name.

ASCII. Standard translation table used to convert characters to their hexadecimal representation and vice versa. See Appendix B for the complete ASCII character set of the PC.

Assembly language. Language that allows programmers to work directly with the machine-language instructions the processor "understands," while providing English-like commands that are easier to work with than the ones and zeroes of machine language.

Asynchronous. Communications over a modem that occur on an irregularly timed basis. See synchronous.

Backup. Copy of the information on a mass-storage device. Protects against the loss or destruction of the original information. Also refers to the process of creating a backup.

Bandwidth. Measure of the information-handling capability of electronic circuitry.

BASIC. Beginners All-purpose Symbolic Instruction Code. A programming language originally designed for teaching, but presently the most widely used language in the microcomputer world.

- Batch processing.** Direction of the operation of a computer from commands that are stored in a file rather than typed at the keyboard.
- Baud.** Measure of the rate at which units of information are transmitted between the computer and another device. The two baud rates commonly used by microcomputers are 300 baud (approximately 30 characters per second), and 1200 baud (approximately 120 characters per second). "Overhead" can lower the effective rate of transmission.
- Bell 103.** Universal data transmission standard for 300-baud modem communications. A Bell 103-type modem can only "speak" to another Bell 103-type modem, but because Bell 103-type modems are so widely used, this is rarely a problem.
- Bell 212A.** Most widely used data transmission standard for 1200-baud modem communications. However, there are other 1200-baud standards that are incompatible with Bell 212A-type modems. A Bell 212A-type modem can only "speak" to another Bell 212A-type modem, so you should make sure that any 1200-baud modem you buy is compatible with the computers you want to communicate with.
- Binary.** Base 2, a numeric notation that has only two digits, 0 and 1.
- BIOS.** Basic Input/Output System. The programs stored in ROM that provide a low-level interface to the devices attached to the PC such as the video screen and keyboard.
- Bit.** Abbreviation for binary digit. A bit is one digit in base 2 (which can only have a value of 0 or 1). The smallest unit of storage in any computer. All information is stored as a series of bits.
- Board.** Any board containing electronic components, such as the system board that contains the "nucleus" of the PC. Also refers to add-on circuit boards, such as a serial port or disk controller.
- Boot.** Process of starting a computer and loading an operating system into memory, usually from disk.
- Buffer.** Portion of memory used as a temporary holding

- area for information that is enroute from one to another.
- Bug.** Error or malfunction in the performance of a program.
- Bus.** Set of common wires used to exchange information between, as well as transmitting power to, the components of a computer, including the processor, memory, and expansion cards.
- Byte.** Group of 8 bits that can have any of 256 different values. The value of a byte can represent a character (see Appendix B) or a number between 0 and 255, and strings of bytes can represent words or larger numbers. A byte is the smallest unit of memory most microcomputers can work with, so memory size is often measured in terms of bytes.
- Card.** Board containing electronic components that can be added to a PC or other computer to provide additional functions or control outside devices.
- Centronics compatible.** Standard parallel interface specification for connecting printers to computers.
- Cluster.** Smallest amount of additional disk space that DOS allocates when it needs to expand a file.
- Code.** Group of program lines or instructions. Can refer to a complete program or a fragment. Also refers to the process of writing programs.
- Command.** Instruction issued to the computer to perform a specific task.
- Compiler.** Program that translates programs written in a high-level language into machine language so they can be run.
- Composite video.** Single video signal that contains the information necessary to create a video picture. See RGB.
- Concatenation.** Process of combining two or more sets of data into one set by placing the beginning of one set directly after the end of the other.
- Constant.** In a BASIC program, a single value that cannot be changed. See variable.
- Controller.** Circuitry that directs the operation of a device attached to the PC.

- Coprocessor.** Second processor that works in conjunction with the computer's main processor in performing special tasks at high speed.
- CPU.** Central Processing Unit. Performs data manipulation and is the central control in a computer. Often used interchangeably with the terms processor and microprocessor.
- CRT.** Cathode Ray Tube. Main display tube in monitors and television sets. Sometimes used to refer to an entire display unit.
- Cursor.** Visual indicator that marks the position on the screen where the next character will appear.
- Database.** Collection of related data organized so it can be expanded, manipulated, recalled, and displayed by a computer program.
- Data set.** Any collection of related information.
- Debug.** Process of detecting and correcting errors in a program.
- Decimal.** Base 10, the familiar notation for representing numeric values with the ten digits 0 through 9.
- Delimiters.** Characters used to mark the beginning and end of a string of characters. Delimiters themselves are not part of the string delimited.
- Digital.** Circuitry that operates with electrical signals which can represent only one of two values, although multiple digital signals can be combined to represent any range of values. See analog.
- DIP.** Dual Inline Package. A standard package for circuits installed inside computers and related peripherals. DIP packages have two rows of parallel "legs" or sockets to accept such "legs."
- Directory.** Index to all files on a disk. A computer-readable directory is stored on each disk to keep track of the contents of that disk.
- DMA.** Direct Memory Access. Circuitry in a computer that can send information to or retrieve information from memory without the information having to go through the central processor.
- DOS.** Disk Operating System. Any operating system that supports the use of disk drives. There are many

DOSes available for the PC including DOS 1.1, 2.0, and 2.1. DOS 2.1 provides complete support for the use of the disk drive.

Double precision. Type of number in BASIC, accurate to 16 decimal digits in the approximate range $10E-38$ to $10E+38$. Requires 8 bytes of storage per number. See integer and single precision.

EBCDIC. Translation table used to convert characters to their machine representation and vice versa on many large computers. The PC, like most microcomputers, uses the ASCII character set (see Appendix B) rather than EBCDIC.

Edit. Make changes to a command, the text of a program, or the text contained in a file.

Enter. Transmit a line of text to the PC (e.g., a command or data to be processed). In general, the Enter key must be pressed at the end of a line of text before the PC will act on it.

Execute. Cause the computer to carry out the actions specified by the instructions in a program. Synonymous with run.

File. Collection of related data stored as a single unit on a cassette or disk and referred to with a single name.

File Allocation Table. (FAT) Information table stored on each disk, indicating where on the disk the information comprising each file is stored. There is an extra copy of the FAT on each disk in case the main FAT is damaged.

Filter. Program that takes input from the standard input, modifies it, and transfers it to the standard output.

Firmware. Usually refers to programs stored in ROM. Named because ROM is more permanent than software but not physically wired like hardware.

Format. Prepare a disk for use with DOS; in this usage, format is equivalent to initialize. Also refers to the arrangement of displayed or printed output in a specific form.

Fragmentation. The degree to which a file is "scattered" on a disk. The more fragmented a file, the slower the process of reading it.

Full duplex. Form of modem communication in which information can be transmitted in both directions simultaneously.

Function. BASIC keyword that calculates a number or string on the basis of other numbers and/or strings.

Glitch. Transient error caused, for example, by a power line surge or phone line interference.

Graphics. On-screen display of detailed images composed of dots.

Half duplex. Form of modem communication in which information can be transmitted in only one direction at any time, so the two communicating devices must "take turns."

Handshaking. Manner in which the flow of data between two devices is controlled.

Hardware. Physical equipment of which a computer is comprised, including the system board, keyboard, display, and add-on cards.

Hexadecimal. (hex) Base 16, a numeric notation with 16 digits, 0 through 9 and A through F.

High-level language. Language that allows programmers to "direct" the computer with easy-to-use English-like instructions, rather than the machine language that the processor understands directly.

I/O. Input/Output. Reception of information by and transmission of information from a computer.

Initialize. Usually describes the preparation of a disk for use by DOS. Often used interchangeably with the term "format."

Input. Acceptance of information by the CPU from an external source such as the keyboard, a disk drive, or modem.

Integer. Whole number. In BASIC, a number with no fractional part in the range -32768 to +32767, and requiring 2 bytes of storage. See single precision and double precision.

Interface. Circuitry that lets the PC connect with an attached device such as a printer, often in the form of an add-on card. Such cards are known as interface cards.

- Interpreter.** Program that executes source programs written in some high-level language without the need for the source program to be compiled and linked as separate steps. See compiler.
- K.** Abbreviation for kilobyte; 1K of memory is 1024 bytes.
- Keyword.** Word recognized by BASIC as one of its reserved set of instructions.
- Kilobyte.** 1024 bytes or 1K bytes. Often written as Kb.
- Language.** Similar to a spoken language, a computer language consists of a vocabulary and set of rules for communicating to the computer the instructions and data necessary for performing tasks. All programs are written in one type of computer language.
- Latch.** Hardware that holds a single value until it is used or replaced.
- LED.** Light-Emitting Diode. An electronic device that emits light, usually a red light which indicates the state of an electronic device or circuit. The light that indicates when the PC's disk drive motor is on is an LED.
- Linker.** Program that takes machine-language files produced by compilers and/or assemblers and combines them into a runnable program file.
- Listing.** Text of a program displayed on the screen or in printed form.
- Load.** Process of transferring the contents of a file into memory.
- M.** Abbreviation for mega, which refers to 1024 times 1024 or 1024K; often used to describe the size of memory, so 1M of memory is 1024K (1,048,576) bytes.
- Machine language.** Language that a processor understands directly. Machine language consists of strings of binary numbers that represent the actual physical operations a processor is designed to execute.
- Mainframe.** A large, multi-user computer typically handling data in 32-bit chunks. See minicomputer.
- Mass storage device.** Device that "holds" information on a long-term basis. Programs and data residing in RAM

can be stored to and retrieved from mass storage devices such as cassette recorders and disk drives.

Megabyte. Approximately one million bytes (1024Kb, or 1,048,576 bytes). Often written as Mb.

Memory. Any device that stores information. Usually refers to the computer's internal memory which is directly and rapidly accessed by the processor and holds programs and data. There are two types of internal memory, RAM and ROM; when neither is specified, memory usually refers to RAM.

Menu. Options displayed on the screen.

Microcomputer. Computer that is much smaller than a mainframe and smaller than a minicomputer, typically handling data in 8-bit chunks, although some microcomputers use 16- or 32-bit data. The boundary between minicomputers and microcomputers is becoming blurred.

Microprocessor. Processor that is contained entirely on a single integrated circuit chip. Often used interchangeably with the terms processor and CPU.

Minicomputer. Computer that is smaller than a mainframe and larger than a microcomputer, typically handling data in 16-bit chunks. The boundary between minicomputers and microcomputers is becoming blurred.

Modem. Abbreviation for MODulator/DEModulator. A device that allows a computer to exchange information with other computers over a telephone line.

Monitor. Display especially designed to show computer-generated text and/or graphics.

Nibble. One-half of a byte or four bits.

Octal. Base 8, a numeric notation in which there are 8 digits, 0 through 7.

Operating system. Software that transforms a computer into a self-contained environment for using programs. Provides the user and programs with easy access to the physical resources of the computer.

Output. Transmission of information from the CPU to another device, such as a disk drive, printer, or modem.

- Parallel.** A means of exchanging data in which all 8 bits of each byte transmitted are sent simultaneously. See serial.
- Parameter.** Value, entered on a command line, that instructs the command to perform a certain action. Parameters can have a user-specified range of values, depending on the action desired.
- Parity bit.** Additional bit sometimes added to a byte, used to check whether the contents of the byte have been transmitted or stored correctly. With odd parity the parity bit is set to 1 or 0, as needed, to make the sum of the bits in the byte odd; with even parity the parity bit is set to 1 or 0, as needed, to make the sum of the bits in the byte even. If the bits in the byte plus the parity bit do not add up to an odd or even number (whichever type of parity is in use) at any time, then the information in the byte is known to have been "corrupted."
- Peripheral.** Device attached to a computer that provides it with additional capabilities (e.g., printer or modem).
- Permanence.** Amount of time that the dots on a CRT remain glowing after having been made to glow by the sweeping electron beam that creates the image. The higher the permanence, the less flicker on the screen, and the less tiring the screen is to view.
- Piping.** Redirection of the standard output of one program to the standard input of another, indicated with the vertical bar character (|).
- Pixel.** Abbreviation for picture element, the smallest unit of area that can be controlled on a screen, printer, or some other display device.
- Port.** Channel through which a computer can send information to and receive information from an external device such as a modem or a printer. There are, in general, two types of ports, parallel and serial.
- Power supply.** Circuitry that provides electricity in the form required by a device. Power supplies for computers usually turn standard 120 volts household alternating current into direct current at one or more lower voltages.

- Print buffer.** Device, placed between a computer and a printer, that accepts text at high speed from the computer, storing it and sending it to the printer at the printer's slower speed. A print buffer "frees" the computer to do other tasks when large files are printed.
- Processor.** "Brain" of a computer, the circuitry that performs data manipulations and calculations and controls the overall functioning of the computer. Often used interchangeably with the terms microprocessor and CPU.
- Program.** Set of instructions to the computer, designed to perform a certain task. Often used interchangeably with the term software.
- Prompt.** Character or string of characters displayed on the screen to indicate that a response should be entered. Also used to describe the process of displaying a prompt.
- Protocol.** Set of rules for transmission of data. The computers on both ends of a communications link must use the same protocol to control the flow of information.
- RAM.** Random Access Memory. Storage locations that can be directly and rapidly accessed by the processor, the contents of which can be changed. RAM is volatile, its contents are retained only as long as the power to the computer remains on.
- RAM disk.** Software that lets DOS use a portion of memory as an ultra-fast simulated disk drive.
- Read.** Process of getting a value or values into the processor from memory, the keyboard, a modem, or another device.
- Read-only.** Storage from which data can be read, but to which data cannot be written so the contents are fixed. ROM is Read Only Memory. Disks can be made read-only with a write-protect tab.
- Redirection.** Feature of DOS that allows the data read from the standard input (usually the keyboard) to come from any file or device, and the data sent to the standard output (usually the screen) to go to a file or device.

- Reserved words.** Set of approximately 200 words, reserved by BASIC for its own use, which cannot be used as variable names. (See Chapter 3 of the *BASICmanual*.)
- Resolution.** Amount of detail in an image that a device is capable of creating. Usually used as a measure of quality for screen displays, indicating how many rows and columns of pixels the screen can display.
- RF modulator.** Abbreviation for Radio Frequency modulator. A device that transforms the standard composite video signal put out by a computer into the form that a television set uses so the computer output can be displayed on a television set.
- RGB video.** Abbreviation for Red-Green-Blue video. Refers to systems that transmit the color information necessary to create a video picture on three separate signals, one for each of these three primary colors of light.
- ROM.** Read Only Memory. Storage locations that can be read directly and rapidly by the processor. The contents of ROM are fixed and, therefore, cannot be changed by the processor.
- Routine.** Series of program lines that work together to produce a specific result. Can be an entire program or a portion of a program that performs a specific task.
- RS-232.** Standard specification for information interchange through a serial port. (RS-232C describes proper electrical connection and voltages.)
- Run.** Cause the computer to carry out the actions specified by the instructions in a program. Synonymous with execute.
- Scrolling.** Move the contents of the entire screen up one line so there is room at the bottom to add a new line.
- Sector.** Basic unit of storage on a disk. The sector size that DOS uses is 512 bytes per sector, with a total of 720 sectors on each disk.
- Separator.** Character that sets apart parameters on command lines. Separators used by DOS include spaces, commas, and semicolons; BASIC most often uses commas.

Serial. Means of exchanging data in which the 8 bits of each byte transmitted are sent one at a time, sequentially. See parallel.

Single precision. Type of number in BASIC, accurate to 6 decimal digits in the approximate range $10E-38$ to $10E+38$. Requires 4 bytes of storage per number. See integer and double precision.

Software. Set of instructions to the computer, designed to perform a certain task. Often used interchangeably with the term "program."

Spooler. Software that "queues" tasks for a slow device, sending them to the device at the speed it can handle, thereby allowing the user to perform other tasks. When used with microcomputers, most often refers to a print spooler which prints large files by using small slices of the computer's time while the user performs other tasks.

Standard input. Device (typically the keyboard) to which programs that run under DOS usually "look" to read data. The user may redirect the standard input so that a program will read from any file or device attached to the PC.

Standard output. Device (typically the display screen) to which programs that run under DOS usually send data to be written. The user may redirect the standard output so that a program will write to any file or device attached to the PC.

Start bit. Bit, used for synchronization, that precedes each byte transmitted asynchronously through a serial port. Automatically added at the source end and removed at the destination end by hardware.

Statement. BASIC keyword that performs a task.

Stop bit. Bit, used for synchronization, that follows each byte transmitted asynchronously through a serial port. One or more stop bits may be used, and both ends of any communications link must use the same number of stop bits. Automatically added at the source and removed at the destination by hardware.

Store. Process of placing a value in memory or on a mass-storage device for later retrieval.

- String.** Sequence of one or more characters. In BASIC, string is a type of constant or variable representing from 0 to 255 characters in a given order.
- Subroutine.** Series of program lines, designed to perform a function. It can be invoked from many places in a program without duplicating the subroutine in each place the function is to be performed. This is accomplished by branching to the subroutine code each time the specific function is required (known as calling the subroutine), then returning to the main body of the program after the subroutine has completed its function.
- Switch.** Optional two-character string entered on a DOS command line which, if present, instructs the command to perform in a certain way. Switches consist of a single character preceded by a slash (/).
- Synchronous.** Events that occur on a regular basis, usually used to describe the manner in which characters are transmitted between computers. Many large computers communicate synchronously but microcomputer-to-microcomputer communications are almost always performed asynchronously.
- System unit.** Box containing the processor and primary circuitry of the PC.
- Track.** Circular magnetic storage area on a disk. PC DOS 2.0 disks have 9 sectors (each storing 512 bytes) on each track, allowing each track to hold 4.5K bytes. There are 40 concentric tracks on each side of a disk, so there are 80 tracks of 4.5K each, for a total of 360K bytes of storage per disk.
- Variable.** Named storage location in BASIC which can hold any one of a range of values and the contents of which can be changed any number of times. See constant.
- VDT.** Video Display Terminal. Any screen and keyboard (plus required support circuitry and a cabinet) used for computer input and output.
- VDU.** Video Display Unit. Any screen (plus required support circuitry and a cabinet) used to display computer output.

Volatile. In computer applications, means not permanent; in particular, refers to the memory-retention characteristics of RAM memory which loses its contents when power is turned off.

Write. Process of sending a value or values from the processor to memory, a printer, a modem, or another device for storage or display.

Write-enable notch. Cutout notch on the side of a disk that must be present before a drive will allow the disk to be written on. Also called the write-protect notch.

Write-protect tab. Adhesive paper or foil that fits over the write-enable notch on a disk so the disk cannot be written on, thereby protecting the contents of the disk from intentional or accidental erasure.

INDEX

A

Absolute addressing, 71-72
 Absolute and Relative Screen Addressing program, 74
 explanation of, 74
 Adapter
 Color/Graphics, 17-24, 33-35, 246
 monochrome, 17, 21, 33-35, 246
 other color/graphics, 22, 24
 Address, 265-266
 Addressing, 71-74
 absolute, 71-72
 relative, 72-74
 Advanced BASIC, 16, 27-30, 33, 49-50
 Alt key, 26, 43, 256-257
 Animation, 108-111, 210-211
 Arcs, 86-88
 Array, character, 154
 Artifacts, 237-240
 ASCII, 270
 Aspect, screen, 88-92
 Aspect ratio, 89-92, 144-145, 175-177
 Assembly language, 280
 Attribute, 208-210

B

Background color, 58-63
 Background Colors program, 62
 explanation of, 61-62
 Backspace key, 26
 BASIC, 16, 29-32
 Advanced, 16, 27-30, 33, 49-50
 Cassette, 49
 Disk, 48-49
 disk, 31-32
 getting started with, 33-50
 BASICA. *See* Advanced BASIC
 BEEP, 219
 Binary, 265-266
 BIOS, 269, 280
 Bit, 265-266
 BLOAD, 249-251
 Block and Half-Block Characters in Text Mode program, 206
 description of, 205

Blockbuster—Ball program, 127
 explanation of, 124-127
 Blockbuster—Bricks program, 123
 explanation of, 122
 Blockbuster—Finished, Commented Version
 program, 129-130
 description of, 129
 Blockbuster—Finished Version
 program, 128
 explanation of, 127-128
 Blockbuster—Paddle program, 124
 explanation of, 123-124
 Blockbuster—Playing Field
 program, 122
 explanation of, 121
 Booting, 28
 Border color, 196-200
 Break key, 26
 BSAVE, 249-251
 BSAVE and BLOAD Statements in Medium-Resolution Mode
 program, 251
 explanation of, 250
 Buffer, 254-256
 Burst, 57-58
 Byte, 265-266
 attribute, 208-210

C

Caps Lock key, 26, 252-254
 Cassette BASIC, 49
 Character array, 154
 Character Generation Package
 Test program, 164-165
 description of, 163
 Character-Generation Package
 program, 166-167
 explanation of, 163, 165, 167-168
 Character(s)
 entering all 255, 256-257
 generation, 151-170
 set, 257-259
 CHKDSK, 30, 32
 CHR\$, 203-204
 CIRCLE, 84-92, 175

Graphics for the IBM PC

CIRCLE Statement in High-Resolution Mode program, 175

description of, 175

CIRCLE Statement program, 86

explanation of, 86

CLS, 65, 200

Colons, 44-45

COLOR, 60-63, 196-200

Color, 241-242

background, 56-63, 196-200

border, 196-200

foreground, 196-200

high-intensity, 61-63

in high-resolution, 237-238

Color Generation by Artifacts program, 240

explanation of, 239-240

Color/Graphics Adapter, 17-24, 33-35, 246

Color graphics screens, 17-22, 33-35

Color selection, medium-resolution 58-63

Color Text Modes program, 23, 199

description of, 199

use of, 22

Colored text, 242-243

Command(s), 39-40

BLOAD, 249-251

BSAVE, 249-251

CHKDSK, 30, 32

DIR, 30

DISKCOPY, 30

FORMAT, 30

KILL, 32

LIST, 34, 36, 38

LOAD, 34

MERGE, 168-169

MODE, 259-260

PSET, 64-68

STEP, 72-73

SYSTEM, 31

Command entry, 42

in lowercase, 29, 38

multiple lines, 43-44

Commas, 56

Comments, 44-46

Composite monitor, 19, 21-23

Concentric Boxes in Text Mode program, 205

description of, 204

Connection box (for tv), 22

Control Caps Lock and Num Lock program, 255

explanation of, 254

Coordinates, 64

Ctrl, 26

Ctrl-Alt-Del, 26, 43

Ctrl-Break, 26, 43

Ctrl-End, 65

Ctrl-Home, 41, 65

Ctrl-Num Lock, 26

Cursor, 40

D

DATA, 6

Debug, 49

Deferred mode, 38-39

DEF SEG, 236-237, 267-269

DEFINT, 211

Determine Display Adapter

Installed program, 247

explanation of, 246

DIR, 30

Direct mode, 37-39

Disk, 27-30

Disk BASIC, 48-49

Disk drive, 27

DISKCOPY, 30

Display. *See also* Screens

connectors, 23

modes, 52

graphics, 52-53

text, 52-53

DOS, 27-28, 30-31

Down arrow, 42

DRAW, 133-150, 176-177

error conditions, 149

variables in, 145-146

equal sign, 145

semicolon, 145

subcommands. *See*

subcommands

substrings, 146-148

summary of, 149-150

DRAW Statement Animation program, 144

description of, 143-144

DRAW Statement in High-Resolution Mode program, 178

explanation of, 177

DRAW Statement Movement

Subcommands program, 139

description of, 138-139

DRAW Statement Substring
 Commands program, 148
 description of, 147
 Dual screens, 17, 33-36
 Dump, screen, 252

E

Edit keys, 41-42
 Editing, 41-42
 8086, 280
 8088, 266-267
 Ellipse, 88-92
 END, 61
 Enter key, 25, 41-42
 Error(s), 32
 Conditions for DRAW, 149
 full disk, 32
 illegal function call, 47-48, 94
 run-time, 47-48
 syntax, 47
 too many files, 32
 ESC key, 26, 43, 65

F

FOR . . . NEXT, 81
 Foreground color, 196-200
 FORMAT, 30
 FORTRAN, 279-280
 Function, 179
 Function(s)
 CHR\$, 203-204
 PEEK, 268-269, 279
 POINT, 75-78, 173, 207
 RND, 224
 SCREEN, 207-210
 Function keys, 26, 39-40, 42-43

G

GET, 103-112, 176
 GET and PUT Statements in High-Resolution Mode program, 177
 description of, 176
 Getting started with your PC, 16-32
 Graphics Display Test program, 36
 explanation of, 36-37
 Graphics screen
 getting onto, 33-35
 text in, 69-71
 Graphing, function, 179-188

H

Hexadecimal, 265-266
 High-intensity, 40, 61-63
 High-intensity colors, 61

High-Resolution Graphics Mode
 Color on RGB Monitors
 program, 238
 description of, 237
 High-resolution mode, 171-178, 232
 color in, 237-238
 versus medium-resolution mode, 177-178
 Home key, 26
 Housekeeping, 114

I

Illegal function call errors, 47-48, 94
 Immediate mode, 37
 Indirect mode, 37-39
 Initializing Character-Generation Package I program, 156
 explanation of, 154-157
 Initializing Character-Generation Package II program, 160-161
 explanation of, 157-158, 161-163
 INKEY\$, 86
 Intensity, 40

K

Kaleidoscope program, 46
 explanation of, 46-47
 KEY OFF, 200, 207
 Keyboard, 25-26, 252-254
 Key Buffer Clearing with INKEY\$ program, 256
 description of, 255-256
 Key(s)
 Alt, 26, 43, 256-257
 Backspace, 26
 Break, 26
 Caps Lock, 26, 252-254
 Ctrl, 26
 Ctrl-Alt-Del, 26, 43
 Ctrl-Break, 26
 Ctrl-Esc, 65
 Ctrl-Home, 41, 65
 Ctrl-Num Lock, 26, 252-254
 down-arrow, 42
 Enter, 25, 41-42
 Esc, 26, 43, 65
 function, 26, 42
 Home, 26
 Num Lock, 26
 numeric keypad, 41
 PrtSc, 252
 right-arrow, 41

Graphics for the IBM PC

Key(s)—cont

- soft, 26, 42-43
- up-arrow, 41

L

- Last Point Referenced, 64, 71-74
- LINE, 79-82, 173-174
- Line numbers, 37-38
- LINE Statement in High-Resolution Mode program, 174
 - description of, 174
- LINE Statement program, 83
 - explanation of, 82
- LIST, 34, 36, 38
- LOAD 34
- LOCATE, 70-71, 191, 200-202

M

- Medium-resolution color selection, 58-63
- Medium-Resolution Graphics Mode Intensity Effects program, 63
 - explanation of, 62-63
- Medium-Resolution Graphics Mode Text program, 72
 - explanation of, 71
- Medium-resolution mode, colors available in, 56-57
 - color selection, 58-63
 - setting up, 57-63
 - versus high-resolution mode, 177-178
- Memory
 - accessing from BASIC, 267-269
 - organization—high-resolution mode, 272-276
 - organization—medium-resolution mode, 276-279
 - organization—text mode, 269-272
- Memory-mapped video, 266-267
- MERGE, 168-169
- MODE, 259-260
- Mode
 - deferred, 38-39
 - high-resolution, 171-178, 232
 - immediate, 37
 - indirect, 37-39
 - program, 37-38
 - screen addressing, 71-74
 - text mode graphics, 189-213, 232-233
- Monitor
 - composite, 19, 21-23
 - artifacting and, 238-240

Monitor—cont

- RGB, 19-22, 171-172
- television set, 19-22
- Monitor Caps Lock and Num Lock program, 253
 - explanation of, 253
- Monochrome, 17-18, 20-21
- Monochrome adapter, 17, 21, 33-35, 246
- Monochrome screen, 17-18, 21, 33-35, 190-191
- Multiple Pages in Text-Mode Animation program, 212-213
 - description of, 210
- Multiple screen pages, 210
- Multiple screens, 190-191

N

- Num Lock key, 26, 252-254
- Numeric keypad, 41

O

Option(s)

- PUT, 105-107
 - AND, 107
 - OR, 106-107
 - PRESET, 106
 - PSET, 106, 110-111
 - XOR, 106, 109
- OUT, 236

P

- PAINT, 93-97, 175-176
- PAINT Statement program, 97
 - explanation of, 96-97
- Palette, 58-63
 - colors, 58-63, 243
 - third (unofficial), 243
 - 0 and 1, 58-63
- Palette 0 and 1 Demonstration program, 61
 - explanation of, 60-61
- Parameter, 55-56
- PC, 16, 27
- PC-DOS, 16, 27
- PEEK, 268-269, 279
- Pie Chart, 98-102
- Pie Chart program, 99
 - explanation of, 98-101
- Picture element, 40
- Pixel, 40, 55
- Plot Specified Function in High-Resolution Mode program, 181-182
 - explanation of, 180, 182-188

POINT, 75-78, 173, 207
 POINT Function in High-Resolution Mode program, 174
 description of, 173
 POINT Function program, 78
 explanation of, 77
 POKE, 268-269, 279
 PRESET, 67-69
 PRINT, 69-71, 191, 202-207
 Print string, 161-162
 Print 255 PC Characters program, 258
 explanation of, 257-258
 Print 255 PC Characters in Color program, 260
 description of, 259
 Program
 clearing, 38
 comments, 44-46
 editing, 41-42
 execution, 37-39
 lines, 43-44
 loading, 34, 39
 mode, 37-39
 numbering, 37-38
 renumbering, 38
 saving, 35, 39
 style, 44-46
 PrtSc key, 252
 PSET, 64-68, 75, 79, 82, 172-173
 PSET and PRESET program, 69
 explanation of, 68-69
 PSET Statement in High-Resolution Mode program, 173
 description of, 173
 PUT, 103-112, 176, 191
 Option(s), 105-107
 AND, 107
 OR, 106-107
 PRESET, 106
 PSET, 106, 110-111
 XOR, 106, 109
 PUT Statement with PSET Option and Border program, 112
 description of, 111
 PUT Statement with PSET Option program, 108
 explanation of, 108
 PUT Statement with XOR Option program, 110
 description of, 109-110
 PUT Statement Options program, 107
 description of, 107

R

Racecar—Car program, 227
 explanation of, 225-227
 Racecar—Finished, Commented Version program, 229-230
 description of, 229
 Racecar—Finished Version program, 228
 explanation of, 227-229
 Racecar—Initial Screen program, 222
 explanation of, 221
 Racecar—Moving Track program, 225
 explanation of, 222-225
 RANDOMIZE, 222-225
 Reference string, 154
 Relative addressing, 72-74
 REM, 45-46
 Resolution, 40, 189-190
 RF modulator, 22
 RGB monitor, 19-22, 171-172
 Right-arrow key, 41
 RND, 224
 RUN 36, 38
 Run-time errors, 47-48

S

Screen aspect, 88-92
 Screen dump, 252
 SCREEN function, 207-210
 SCREEN Function program, 211
 explanation of, 209-210
 SCREEN statement, 57-58, 192-195
 Screen(s)
 alignment, 259-260
 checking the, 248
 color graphics, 17-22
 dual, 17, 33-36
 monochrome, 190-191
 multiple, 190-191
 Scrolling, 70
 Scroll Window in Medium-Resolution Mode program, 262
 explanation of, 261-262
 Scroll window, variable, 260-262
 Semicolons, 70, 145, 205-207
 Shift state, 252-254
 16 Background Colors program, 62
 explanation of, 61-62
 Soft keys, 26, 38, 42-43
 Software, 27-28

Graphics for the IBM PC

Statement, 39-40

- BEEP, 219
- CIRCLE, 84-92, 175
- CLS, 65, 200
- COLOR, 60-63, 196-200
- DEF SEG, 236-237, 267-269
- DEFINT, 211
- DRAW, 133-150, 176-177
- END, 61
- FOR . . . NEXT, 60, 81
- GET, 103-112, 176
- KEY OFF, 200, 207
- LINE, 79-82, 173-174
- LOCATE, 70-71, 191, 200-202
- OUT, 236
- PAINT, 93-97, 175-176
- POKE, 268-269, 279
- PRESET, 67-69
- PRINT, 69-71, 191, 202-207
- PSET, 64-68, 75, 79, 82, 172-173
- PUT, 103-112, 176, 191
- RANDOMIZE, 222-225
- REM, 45-46
- SCREEN, 57-58, 191-195
- STOP, 61
- WIDTH, 195-196

STEP, 72-73

STOP, 61

Subcommands, DRAW

- A, 135, 142-145
- B, 135, 139
- C, 135, 140-141
- D, 135, 137
- E, 135, 137-138
- F, 135, 137-138
- G, 135, 137-138
- H, 135, 137-138
- L, 135, 137
- M, 133-135
- N, 135, 140
- R, 135, 137
- S, 135, 141-142
- U, 135, 137
- X, 135, 147-149

Subroutine(s), 153

Switch program, 35

explanation of, 34-35, 244, 247

Switch with error message program, 248

description of, 247-248

Syntax errors, 47

SYSTEM, 31

T

Television set, 19-22

connection box for, 22

Text Character Set program, 207

description of, 205

Text in the graphics screen, 69-71

Text mode graphics, 189-213,
232-233

Text-Mode Color Shading program, 206

description of, 204

Text-Mode Graphics program, 203

description of, 202

Text, nature of, 151-153

Three Color Palettes program, 244

description of, 243

Tutorial, 52-232

introduction to, 52-54

summary of, 232-234

Two-Adapter Graphics, 245

explanation of, 245

U

Up-arrow key, 41

V

Variable

INKEY\$, 86

Variable Parameters to DRAW program, 146

description of, 146

Varying Aspect Ratio with CIRCLE Statement program, 91

explanation of, 90-91

Video

memory-mapped, 266-267

W

Wedges, 86-88

WIDTH, 195-196

SCREEN $0,0,0 = 80 \text{ COL. T x TP}$

$0,0,1 = ?$

$0,1,0 = ?$

$0,1,1 = ?$

$1,1,1 = \text{ILLEGAL FUNCTION}$

$1,1,0 = \text{N.G.}$

$1,0,0 = \text{N.G.}$

$0,0,0 =$

$2,0,0 \text{ N.G.}$

$0,0,0$

SCREENS $1,0,0 + 0,0,0$

Graphics for the IBM PC

Some of the many features offered by this comprehensive book about IBM graphics programming include:

- A tutorial approach to graphics
- A fun way to broaden your BASIC knowledge
- Extensive, well thought out example programs
- A description of IBM graphics and directions for putting those graphics to use
- How to design a video game
- How to program graphic displays
- Valuable information gleaned from the experience of accomplished game authors

Graphics for the IBM PC is intended for the nonspecialist in graphics, or computers in general. It is designed for home users who want to learn graphics and business users who desire to produce charts, graphs, and slides with ease.

Howard W. Sams & Co., Inc.

4300 West 62nd Street, Indianapolis, Indiana 46268 U.S.A.

\$14.95/22191

ISBN: 0-672-22191-8